TURING 图灵程序设计丛书

Node: Up and Running



Node

即学即用

[英] *Tom Hughes-Croucher & Mike Wilson* 著

郑达韡 译

O'REILLY®

人民邮电出版社
POSTS & TELECOM PRESS

作者介绍

Tom Hughes-Croucher

程序员，同时也是技术布道师。他曾先后效力于许多响当当的大公司，或与他们保持有合作关系，如雅虎、NASA、Tesco、沃尔玛、MySpace、Three Telecom以及UK Channel 4等。Tom向万维网联盟（W3C）和英国国标准协会（BSI）提交了多项网络标准提案。

Mike Wilson

程序员，系统架构师和管理员。曾与许多世界一流公司开展过合作，包括迪士尼、微软和麦当劳。他有多年网络开发经验，从小企业网站到百万用户在线的大型MMO服务器集群，他都曾设计并构建过。在闲暇时间，Mike会更新他的个人博客（http://www.alwaysgetbetter.com），在论坛上发表文章，以及尝试新的框架和软件。

译者介绍

郑达韡

技术爱好者，热衷于编写Linux服务器端程序。自从接触Node，便爱不释手。翻译此书也是作为对开源社区的一点贡献，希望本书能够帮助中国开发者，并且吸引更多人来使用Node。个人主页：http://zdwalter.info。

TURING 图灵程序设计丛书

Node即学即用

Node: Up and Running, Scalable Server-Side Code with JavaScript

[英] Tom Hughes-Croucher
Mike Wilson 著
郑达韡 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　京

图书在版编目（C I P）数据

Node即学即用 / (英) 休斯-克劳奇
(Hughes-Croucher,T.)，(英) 威尔逊 (Wilson,M.)著;
郑达韡译. -- 北京 : 人民邮电出版社, 2013.1
（图灵程序设计丛书）
书名原文: Node: Up and Running, Scalable
Server-Side Code with JavaScript
ISBN 978-7-115-30618-0

Ⅰ. ①N… Ⅱ. ①休… ②威… ③郑… Ⅲ. ①
JAVA语言—程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2013)第005230号

内 容 提 要

本书讲解如何用Node构建可扩展因特网应用，是全面的实用指南，除了详细介绍Node提供的API外，还用大量篇幅介绍了服务器事件驱动开发的重要概念。内容涉及跨服务器的并发连接、非阻塞I/O和事件驱动的编程、如何支持各种数据库和数据存储工具、Node API的使用示例等。

本书适合对JavaScript及编程有一定程度了解的读者阅读。

图灵程序设计丛书

Node即学即用

◆ 著　　[英] Tom Hughes-Croucher Mike Wilson
译　　郑达韡
责任编辑　朱　巍
执行编辑　李　瑛

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街14号
邮编　100061　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：12.25
字数：242千字　　　　2013年1月第1版
印数：1－3 500册　　　　2013年1月北京第1次印刷

著作权合同登记号　图字：01-2012-4261号

ISBN 978-7-115-30618-0

定价：39.00元

读者服务热线：(010)51095186转604　印装质量热线：(010)67129223

反盗版热线：(010)67171154

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会聚集了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——Wired

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——CRN

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

Ryan Dahl序

2008 年，我在寻找一个新的编程平台来做网站。我并不是想要一门新的语言，实际上，语言自身的细节对我来说并不重要。我真正关心的是，该语言能否提供先进的推送功能并集成到网站中来，就像我在 Gmail 中看到的那样——能够从服务器端把数据主动推送给用户，而不是采用不断轮询拉取数据的方式。现有的平台都把服务器作为接受请求然后返回相应内容的设备。要把事件推送到浏览器，平台需要能够持续处理大量打开的网络连接，而这其中有许多连接其实是空闲的。

Google 在 2008 年年末推出了 Chrome 浏览器和崭新的 JavaScript 引擎 V8。这是一个为了更快的 Web 体验而专门制作的更快的 avaScript 引擎，V8 让 Web 应用大大提速了。突然之间，Google、Apple、Mozilla 和微软之间的 JavaScript 军备竞赛就开始了。再加上 Doug Crockford 的 *JavaScript: The Good Parts* 一书的面世，把 JavaScript 从一门人人轻视的语言一下变成了重要的语言。

于是，我有了个主意：JavaScript 结合非阻塞 socket！因为 JavaScript 并没有现成的 socket 库，所以我可以勇做第一人，来推介这个崭新且大有前途的接口。只要把 V8 接上我的非阻塞 C 代码，我就能把它完成。我终止了当时承接的工作，开始全力实现这个想法。当我编写好并发布了最初的版本后，立刻就有用户开始反馈 bug，然后我开始不停地处理这些 bug，就这样，不知不觉过去了 3 年。

实践证明，JavaScript 与非阻塞 socket 配合得相当完美。开始我并不敢肯定这一点，但闭包让所有事情变得可能。只需要简单的几行 JavaScript 代码，就可以构建出非常复杂的非阻塞服务器。我最初还担心，系统会过于小众，但很快我就放心了，因为世界各地的黑客们纷纷开始为其编写程序库。唯一的事件循环队列和纯粹的非阻塞接口让程序库不必增加昂贵的线程，就能添加越来越多的复杂功能。

在 Node 中，用户会发现系统在默认情况下就能很好地扩展。因为其核心系统做出

的选择是，不允许系统中的任何部分做出太坏的事情来（比如堵塞当前线程），所以整体性能也不会太差。如果以能够处理的流量作为计量，Node 的方法要比传统的阻塞式操作好上一个数量级。

现在，Node 已经在全球被众多公司所使用，包括创业公司、Voxer、Uber，以及沃尔玛、微软这样的知名公司。可以说，每天通过 Node 处理的请求数以亿计。随着越来越多的人参与到本项目中来，可用的第三方模块和扩展增长迅猛，而且质量也不断提升。虽然我曾建议将 Node 用于关键任务应用，但现在，即便是要求最苛刻的服务器系统，我也会热诚地推荐使用 Node。

本书探讨了 Node 及许多第三方模块，并给出了指导练习，旨在带你深入浅出地了解 Node。通过学习本书，你不但能够熟悉 JavaScript 的基本操作，还能逐渐开始构建复杂、交互式的网站。如果你曾经使用过其他服务器端 Web 框架，你会震惊于用 Node 这么容易就能编写一个服务器！

——Ryan Dahl，Node.js 的创建者

Brendan Eich序

1995 年 4 月，我加入了 Netscape 公司，负责“把 Scheme 添加到浏览器里”。一两个月后，这个任务却演变成了“创造一门看起来像 Java 的脚本语言”。更糟糕的是，当时正在商议把 Java 添加到 Netscape 里，所以 Netscape 的一部分人对是否需要一门“第二语言”表示怀疑。同时，另外一部分人想要的是类似 PHP 的东西，也就是为公司计划发布的服务器产品 LiveWire 写的一门 HTML 模板语言。

于是，在 1995 年 5 月，我用 10 天时间开发了 Mocha 原型（代码名称是 Marc Andreessen 挑选的）。当时，Marc、Rick Schell（Netscape 的工程副总裁）和 Sun 公司的 Bill Joy 这几位高层管理者都支持我继续做下去，以消除人们对 Java 之后“第二语言”的怀疑。（极具讽刺的是，Java 几乎已在浏览器世界里绝迹了，而 JavaScript 则成为 Web 客户端的主导。）

为了消除一切疑虑，我需要在 10 天内拿出一个能演示的原型。当时我日以继夜地工作，结果引入了一些设计语言的错误（其中一些重复了 LISP 演变过程中的设计错误），但最终还是赶在期限前完成了演示。

人们很惊讶，我竟然用不到两周的时间就完成了一门语言的编译器和运行环境。其实自从大三那年由物理专业转到数学 / 计算机科学专业起，我已经积累了十多年的经验。我一直很喜欢形式语言和自动机理论，并出于兴趣编写了自己的语言解析器和解析器生成器。在 Silicon Graphics 的时候，我编写的网络监控工具包含了包头匹配、协议描述语言和编译器。此外，我还是 C 和 Unix 的忠实粉丝。所以，弄出 Mocha 只不过是一件需要持续工作与专注的事情。

1995 年秋天，Netscape 市场部把 Mocha 改名为 LiveScript，好让它和服务器产品 LiveWire 的名字相匹配。1995 年 12 月初，Netscape 和 Sun 最终签订一份商标使用许可协议，由创始人 Bill Joy 代表 Sun 公司签字生效，LiveScript 正式改名为 JavaScript（JS）。

因为有 LiveWire 服务器的计划，我在头 10 天里实现了一个字节码编译器和解释器，同时还有反编译器和运行时程序（内置我们今天熟悉的 JS 对象和函数：Object、Array、Function 等）。对于一个小巧的客户端脚本来说，字节码有点大材小用了，因为 LiveWire 产品里包含了一种特性，即能够保存编译好的字节码以供服务器端应用更快地启动。

最终，与 Netscape 其他大部分业务一样，Netscape 的服务器端 JavaScript 产品也失败了，因为微软把 IE 浏览器绑定在 Windows 里，并进入了 Netscape 原本想开拓的浏览器之外的服务器市场。而绑定在 Windows 里的 IE 是免费的，因此商业用户不再需要单独购买付费浏览器证书了。

尽管 LiveWire 失败了，但早在 1995 年我们就已经看到端到端 JavaScript 编程的吸引力。用户也看到了这个趋势，但这段历史只有一小部分人知道。今天，Node.js 避开了 LiveWire 当年的致命错误：把堵塞式输入 / 输出包含在内，并在服务器端使用多进程模型，因而可扩展性并不是很好。

2009 年的 JSConf EU 大会上，Ryan 展示了 Node.js。我为能够了解 Node 而感到欣慰，也很高兴它能够很好地实现彻底使用 JavaScript 的愿景，特别是它能从底层构建起整个非阻塞 I/O 系统。Ryan 和其他核心成员在保持内核精致方面做得很好。Isaac 及所有模块所有者共同构建起的优秀模块系统分担了内核的压力，所以它不会太过臃肿。此外，围绕 Node 代码成长起来的社区也很出色。

结果，这就诞生了一个有趣高效的系统，它不仅能够构建服务器端，而且能够适应日益提高的产能，还可以很方便地进行 JavaScript 客户端程序开发，并能够促进代码重用和进化。如果没有 Node，JavaScript 将只能绑定在 Web 客户端上，其备受指责的文档对象模型以及其他一些历史遗留问题将会日益突出。Node 帮助 JavaScript 摆脱了客户端的限制。

本书很好地诠释了 Node 的精髓，并讲述了如何用它构建交互式网络应用和网站。Node 棒极了，而本书就是关于 Node 的很好的指南，请尽情享受阅读的乐趣吧！

——Brendan Eich，JavaScript 的创建者

前言

介绍

Node.js 正迅速成为 Web 开发社区里最有影响力的技术。本书的目标是让开发人员有效地了解如何入手试用 Node。

本书读者应该对 JavaScript 及编程有一定程度的了解。除了详细介绍 Node 提供的 API 外，我们还将花大量篇幅来介绍服务器事件驱动开发的重要概念。

通过阅读本书，你不但能够了解 Node 平台本身，还能掌握 Node 为快速高效地构建高扩展性网站和服务所提供的多个重要模块。

排版约定

本书使用了下列排版约定。

- 楷体
  表示新术语。

- 等宽字体
  表示程序片段，也表示在正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

- 加粗等宽字体
  表示用户的输入。

这个图标表明提示、建议或一般注记。

这个图标表示警告或警示。

使用代码示例

本书用于帮助你完成工作。通常，你可以在程序或文档中使用本书提供的代码。除非你重新发布我们的大量代码，否则不需要联系我们来获得许可。比如，在程序中使用本书代码的一些片段是无需我们许可的，但是出售或再分发 O’Reilly 的图书示例光盘显然是需要授权的。引用本书或引用示例代码来回答问题是不需要授权的，但是将本书的大量示例代码整合到你自己的产品文档必须得到授权。

我们希望你在使用时声明引用信息，但不强求。引用信息通常包括书名、作者、出版社和 ISBN。例如：“Node: Up and Running by Tom Hughes-Croucher and Mike Wilson (O’Reilly). Copyright 2012 Tom Hughes-Croucher and Mike Wilson, 978-1-449-39858-3.”

如果你认为对示例代码的使用需要授权，请通过邮箱 permissions@oreilly.com 联系我们。

Safari®在线图书

在线图书是应需而变的数字图书馆。它能够让你非常轻松地搜索 7500 多种技术性和创新性参考书以及视频，以便快速地找到需要的答案。

订阅后就可以访问在线图书馆内的所有页面和视频。可以在手机或其他移动设备上阅读，还能在新书上市之前抢先阅读，也能够看到还在创作中的书稿并向作者反馈意见。复制粘贴代码示例、放入收藏夹、下载部分章节、标记关键点、做笔记甚至打印页面等有用的功能可以节省大量时间。

这本书英文版也在其中。欲访问本书的英文版电子版，或者由 O’Reilly 或其他出版社出版的相关图书，请到 http://my.safaribooksonline.com 免费注册。

我们的联系方式

请将有关此书的意见及问题发给出版商：

美国：

O’Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）

奥莱利技术咨询（北京）有限公司

本书有一个 Web 页面，上面列出了勘误表、一些实例以及所有的附加信息。可以通过以下链接来访问这个页面：

http://oreil.ly/node_upandrunning

为本书提意见或者询问一些技术性问题，可以向以下地址发送邮件：

bookquestions@oreilly.com

更多与书籍、会议、资源中心以及 O’Reilly 网络有关的问题，都请参见 O’Reilly 的网站：

http://www.oreilly.com

致谢

http://www.oreilly.com

感谢我的编辑们。Simon，这是一个漫长的项目，感谢你天天陪伴着我。Andy，你对细节的专注让我印象深刻。

感谢 Carols，你写作的动力和能力让我羡慕，你为我带来了许多灵感。

感谢 Nicole 和 Sean，是你们帮我把握方向，让我保持进度。

感谢 Ryan 和 Isaac，你们像教育孩子那样耐心，不停地回答我那些无休止的傻问题。

感谢 Rosemarie，没有你，我不可能有今天的成绩。

感谢我的朋友们，特别是 Yta、Emily、Eric、Gris、Sarah、Allan、Harold、Daniella 和 Hipster Ariel，感谢你们听我发牢骚。此外，还要感谢无数给我鼓励、建议和反馈的人，没有你们，我无法完成此书。

感谢读者朋友，谢谢你们购买并阅读这本书，谢谢你们信任我。

——Tom

目录

第一部分　基础入门

第二部分　API 和常用模块

第一部分

基础入门

第 1 章

Node.js简介

Node 功能强大，特别是它能在浏览器以外运行 JavaScript。本书将阐述这一功能为何如此重要，以及使用 Node 的好处。首先来概述一下这些特性。

很多人将 JavaScript 用在前端网站应用开发上。Node.js 将这一流行编程语言扩展到了更多的领域，特别是后端网站服务器开发。Node 有几个重要的特性值得我们关注。

Node 是对高性能 V8 引擎的封装（V8 是 Google Chrome 浏览器的 JavaScript 引擎），通过提供一系列优化的 API 类库，使 V8 在浏览器之外依然能高效运行。比如，在服务器端开发程序常常需要处理二进制文件，JavaScript 语言本身对此支持得不好，因此 V8 也如此，而 Node 的 `Buffer` 类库提供了轻松操作二进制数据的方法。使用 Node，除了可以直接操作 V8 的 JavaScript 运行时状态，还能在开发上得到更多益处。

Node 的一大特性是对高性能的追求。首先，V8 采用了编译领域的一些最新技术，使得用 JavaScript 等高级语言编写的代码在运行效率上能够接近用 C 等底层语言编写的代码，并且开发成本有所降低。

其次，Node 利用了 JavaScript 的事件驱动（event-driven）特性来构建高度可扩展的服务器程序。Node 采用了*事件循环*（event loop）架构，让开发高效的服务器程序变得简单和安全。对比其他构建高性能服务器的架构，Node 既保证了性能，又降低了开发难度。这是一个极其重要的特性。大家都知道开发多线程并行程序很困难，而且非常容易出错。Node 却巧妙地回避了这一难题，并且保持着令人惊讶的高性能。当然，任何方法都存在利弊得失，在后续章节中，将会详细讨论 Node 在这其

中是如何取舍的。

Node 提供了一系列“非阻塞”函数库来支持事件循环特性。比如，把文件系统或数据库操作封装成事件驱动形式的函数接口。当对文件系统发起请求时，程序不需要闲置等待硬盘把文件读取出来。就像在浏览器中 `onclick` 事件被触发后会自动调用代码一样，非阻塞函数会在它获得文件内容后通知 Node 中的程序。这种方式让访问慢资源变得简单可扩展，这对 JavaScript 程序员来说可谓驾轻就熟，甚至普通人也很容易掌握。

Node 的强大特性还包括能在服务器端运行 JavaScript，尽管这样的特性并非 Node 所独有。如果想在主流浏览器上运行自己的应用，我们除了 JavaScript 之外没有什么其他选择。那么要想同一份代码在浏览器客户端和服务器间共享，也只能选择 JavaScript。现在出现了越来越多用 JavaScript 编写的复杂网页应用（如 Gmail），如果能把越多的代码共享到服务器上运行，那么开发的成本也会越低。Node 为服务器端共享网页的 JavaScript 代码铺平了道路，这是 PHP、Java、Ruby 或 Python 等其他编程语言无法提供的。虽然也有其他平台提供了在服务器端使用 JavaScript 的手段，但 Node 已经先声夺人，迅速成为这个领域的主流平台。

除了可以用 Node 现有的库来构建应用外，开发者也可以轻松为其扩展新的库，这实在令人欣喜。正因为 Node 很容易扩展，在 Node 项目对外发布后，其社区便迅速涌现出大量扩展库。其中许多是连接数据库或其他软件的驱动接口，还有相当一部分是独立有用的软件。

Node 社区也是非常值得称赞的。虽然其社区非常年轻，但已经罕见地受到许多开发者的热情关注。初学者和专家们都聚集在此项目上，使用它并反馈贡献到社区中，致力于把 Node 社区建设成每个人都能够在其中快乐地探索、分享知识并获得支持的地方。

1.1 安装Node.js

安装 Node.js 是极其简单的事情。Node 能够运行在 Windows、Linux、Mac，以及 Solaris 和 BSD 等其他 POSIX 系统上。Node.js 能够在以下两个地址获得：项目官方主页（http://nodejs.org）和 GitHub 代码库（http://github.com/joyent/node）。你可以优先选择 Node 主页上提供的稳定发布版。包含最新特性的版本托管在 GitHub 上，供核心开发团队使用。任何人想获得一份拷贝也能从 GitHub 上下载。虽然这些新特性通常很炫，但它们没有稳定版本那么可靠。

让我们从安装 Node.js 开始。首先要从 Node 主页下载最新发布的版本。在 Node 主

页上，找到下载的链接。本书印刷时的稳定发布版本是 0.6.13[1]。Node 主页提供了 Windows 和 Mac 的安装程序，以及源代码包。如果你在使用 Linux，可以选择从源代码安装，也可以使用常见的包管理程序（apt-get、yum 等）。

> Node.js 版本号依照 C 的习惯：主版本 . 次版本 . 补丁。稳定版本的次版本号是偶数，开发版本的次版本号是奇数。虽然不知道 Node 什么时候会到达 1.0 版本，但可以认定是在 Windows 和 Unix 版本合并成一个版本同时发布时。

如果你使用安装包，可以直接跳到 1.2 节。若你采用源代码安装，需要首先进行解压。使用 `tar` 命令，带上 `xzf` 参数。`x` 参数表示解压（而不是压缩），`z` 参数告诉 `tar` 用 GZIP 算法进行解压，`f` 表示根据最后一个参数的文件名来解压（见例 1-1）。

例 1-1　代码解压

```
enki:Downloads $ tar xzf node-v0.6.6.tar.gz
enki:Downloads $ cd node-v0.6.6
enki:node-v0.6.6 $ ls
AUTHORS        Makefile         common.gypi      doc          test
BSDmakefile    Makefile-gyp     configure        lib          tools
ChangeLog      README.md        configure-gyp    node.gyp     vcbuild.bat
LICENSE        benchmark        deps             src          wscript
enki:node-v0.6.6 $
```

下一步是根据你的系统进行配置。Node.js 安装采用 configure/make 方法。`configure` 程序将扫描你的系统，查找 Node 依赖库的路径。Node 通常需要很少的依赖库。安装需要 Python 2.4 或更高版本，如果你想使用传输层安全（TLS）或加密（如 SHA1），Node 将需要 OpenSSL 开发库。运行 `configure` 程序将提示你缺少哪些依赖库（参见例 1-2）。

例 1-2　Node 安装的配置

```
enki:node-v0.6.6 $ ./configure
Checking for program g++ or c++          : /usr/bin/g++
Checking for program cpp                 : /usr/bin/cpp
Checking for program ar                  : /usr/bin/ar
Checking for program ranlib              : /usr/bin/ranlib
Checking for g++                         : ok
Checking for program gcc or cc           : /usr/bin/gcc
Checking for gcc                         : ok
Checking for library dl                  : yes
Checking for openssl                     : not found
Checking for function SSL_library_init   : yes
Checking for header openssl/crypto.h     : yes
Checking for library util                : yes
```

注 1：翻译本书时，版本已经是 0.8.1 了。——译者注

```
Checking for library rt                  : not found
Checking for fdatasync(2) with c++       : no
'configure' finished successfully (0.991s)
enki:node-v0.6.6 $
```

接着是运行 make 来编译项目（例 1-3）。这将在我们一直使用的源代码文件夹下编译出可执行的二进制文件。Node 会在编译过程中列出当前进行到第几步，以便查看。

例 1-3　运行 make 来编译

```
enki:node-v0.6.6 $ make
Waf: Entering directory '/Users/sh1mmer/Downloads/node-v0.6.6/out'
DEST_OS: darwin
DEST_CPU: x64
Parallel Jobs: 1
Product type: program
[ 1/35] copy: src/node_config.h.in -> out/Release/src/node_config.h
[ 2/35] cc: deps/http_parser/http_parser.c -> out/Release/deps/http_
parser/http_parser_3.o
/usr/bin/gcc -rdynamic -pthread -arch x86_64 -g -O3 -DHAVE_OPENSSL=1 -D_
LARGEFILE_SOURCE ...
[ 3/35] src/node_natives.h: src/node.js lib/dgram.js lib/console.js lib/
buffer.js ...
[ 4/35] uv: deps/uv/include/uv.h -> out/Release/deps/uv/uv.a

...

f: Leaving directory '/Users/sh1mmer/Downloads/node-v0.6.6/out'
'build' finished successfully (2m53.573s)
-rwxr-xr-x  1 sh1mmer  staff   6.8M Jan  3 21:56 out/Release/node
enki:node-v0.6.6 $
```

最后一步是用 make install 来安装。首先，例 1-4 演示了如何为系统下全部用户安装 Node 程序。这需要你有 root 账户或者有运行 sudo 的权限。

例 1-4　为系统下全部用户安装 Node

```
enki:node-v0.6.6 $ sudo make install
Password:
Waf: Entering directory '/Users/sh1mmer/Downloads/node-v0.6.6/out'
DEST_OS: darwin
DEST_CPU: x64
Parallel Jobs: 1
Product type: program
* installing deps/uv/include/ares.h as /usr/local/include/node/ares.h
* installing deps/uv/include/ares_version.h as /usr/local/include/node/
  ares_version.h
* installing deps/uv/include/uv.h as /usr/local/include/node/uv.h

...

* installing out/Release/src/node_config.h as /usr/local/include/node/
```

```
node_config.h
Waf: Leaving directory '/Users/sh1mmer/Downloads/node-v0.6.6/out'
'install' finished successfully (0.915s)
enki:node-v0.6.6 $
```

如果你想只安装到本地用户，或是不使用 sudo 命令，需要在运行 configure 的时候加上 --prefix 参数，指定路径来代替默认安装（例 1-5）。

例 1-5　安装到本地用户

```
enki:node-v0.6.6 $ mkdir ~/local
enki:node-v0.6.6 $ ./configure --prefix=~/local
Checking for program g++ or c++          : /usr/bin/g++
Checking for program cpp                 : /usr/bin/cpp

...

'configure' finished successfully (0.501s)
enki:node-v0.6.6 $ make && make install
Waf: Entering directory '/Users/sh1mmer/Downloads/node-v0.6.6/out'
DEST_OS: darwin
DEST_CPU: x64

...

* installing out/Release/node as /Users/sh1mmer/local/bin/node
* installing out/Release/src/node_config.h as /Users/sh1mmer/local/
include/node/...
Waf: Leaving directory '/Users/sh1mmer/Downloads/node-v0.6.6/out'
'install' finished successfully (0.747s)
enki:node-v0.6.6 $
```

1.2　开始写代码

本节将介绍一些 Node 开发的基础内容，为进一步学习做准备。

1.2.1　Node REPL

Node 是服务器程序，人们常常难以理解它也有和 Perl、Python、Ruby 一样的运行时环境。所以通常我们称 Node.js 为“服务器端的 JavaScript”，但这不能完全描述 Node.js 本身。了解 Node.js 的最佳方法是使用其提供的 REPL 模式（Read-Evaluate-Print-Loop，输入 – 求值 – 输出 – 循环），即交互式命令行解析器，它非常适合检验和学习 Node.js。你可以在 Node 命令行解析器中试验本书提供的代码片段。此外，因为 Node 是对 V8 的封装，所以 Node 命令行解析器也是用来轻松测试 JavaScript 的理想方法。同时，当你想运行一个 Node 程序时，可以用任何你喜爱的文本编辑器写好并保存成文件，然后运行 node　filename.js。命令行解析器是极佳的学习和探索工具，但我们不会将其用在产品程序中。

让我们启动 Node 命令行解析器，来个热身，试验一下 JavaScript 吧（参见例 1-6）。在你的系统中打开命令行终端。我正使用 Mac 系统的自定义命令行环境，所以你的系统提示可能会有所不同，但使用的命令应该是一样的。

例 1-6　启动 Node 命令行解析器并尝试测试 JavaScript

```
$Enki:~ $ node
> 3 > 2 > 1
false
> true == 1
true
> true === 1
false
```

> 第一行代码返回的结果为 `false`。这个例子来自一个收集 JavaScript 诡异和奇特特性的网站 http://wtfjs.com。

拥有一个实时的开发环境，你就有了非常好的学习工具，但你还需要了解 Node 解析器的一些有用的功能，才能更好地使用它。它提供了以点号（.）开头的元命令。如 `.help` 会显示帮助菜单，`.clear` 会清除当前运行的内容，`.exit` 将退出 Node 解析器（见例 1-7）。其中最有用的命令是 `.clear`，它会清除内存中任何变量或闭包，而不需要重启解析器。

例 1-7　使用 Node 解析器中的元命令

```
> console.log('Hello World');
Hello World
> .help
.clear  Break, and also clear the local context.
.exit   Exit the prompt
.help   Show repl options
> .clear
Clearing context...
> .exit
Enki:~ $
```

使用解析器时，输入变量的名称就会在终端上显示其内容。Node 会尝试智能地显示复杂对象，比如通过描述来反映对象的内部构造，而不是简单地将其当做普通对象来显示（见例 1-8）。主要的例外是显示函数，并非解析器无法显示函数内容，而是因为函数通常都很长，如果解析器把函数都展开，很可能会导致刷屏。

例 1-8　解析器设置并显示对象

```
Enki:~ $ node
> myObj = {};
{}
> myObj.list = ["a", "b", "c"];
[ 'a', 'b', 'c' ]
```

```
> myObj.doThat = function(first, second, third) { console.log(first); };
[Function]
> myObj
{ list: [ 'a', 'b', 'c' ]
, doThat: [Function]
}
>
```

1.2.2 编写首个服务器程序

命令行解析器是我们学习和试验的好工具，而 Node.js 最主要的应用是服务器程序。设计 Node.js 的一个主要目的是提供高度可扩展的服务器环境。这是我们在本章开篇介绍过的 Node 和 V8 引擎有所区别的地方。Node 除了用 V8 引擎来解析 JavaScript 外，还提供了高度优化的应用库，用来提高服务器效率。比如说，HTTP 模块是专为快速非阻塞式 HTTP 服务器而用 C 重新编写的。让我们看一下 Node 采用 HTTP 服务器的“Hello World”经典例子（例 1-9）。

例 1-9 “Hello World”Node.js Web 服务器

```
var http = require('http');
http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World\n');
}).listen(8124, "127.0.0.1");
console.log('Server running at http://127.0.0.1:8124/');
```

这个示例代码首先通过 `require` 方法把 HTTP 库包含到程序中来。有许多语言都有包含其他库这一方法，Node 用的是 CommonJS 模块风格。Node 模块将在第 8 章详细介绍，当前需要了解的是，HTTP 库所具有的功能已经赋给了 `http` 对象。

下一步，我们需要一个 HTTP 服务器。PHP 等其他语言需要在类似 Apache 这样的服务器中运行，而 Node 和它们不同，因为 Node 本身就是 Web 服务器。但这同样意味着我们需要先创建该服务器。下一行代码调用 HTTP 模块的一个工厂模式方法（`createServer`）来创建新的 HTTP 服务器。新创建的 HTTP 服务器并没有赋值给任何变量，它只会成为存活在全局范围内的匿名对象。我们可以通过链式调用来初始化服务器，并告诉它监听在 8124 端口。

当调用 `createServer` 的时候，我们传了一个匿名函数作为参数。此函数绑定在新创建服务器的事件监听器上进行 `request` 事件处理。消息事件是 JavaScript 和 Node 的核心。在这个例子中，每当一个新的访问请求到达 Web 服务器，它都将调用我们指定的函数方法来处理。我们称这类方法为回调（callback）。因为每当一个事件发生时，我们将回调监听此事件的所有函数。

一个很恰当的类比是，你从书店预订一本书，等书到货时，书店会“回调”通知你去取。

例子中的回调函数有两个参数，一个是请求的对象（req），一个是响应的对象（res）。在回调函数中，我们调用了 res 对象的几个方法，这将修改响应结果。例 1-9 没有使用 req 对象，但你通常会需要同时使用请求和响应对象。

首先我们必须调用 res.writeHead 方法来设置 HTTP 响应头，否则就不能返回真实内容给客户端。我们设置状态代码为 200（表示 HTTP 状态代码“200 OK”），并且传入一段 HTTP 头描述。在本例中，我们只指定了 Content-type。

在完成了 HTTP 头后，我们可以写入 HTTP 正文。在本例中，我们用一个方法来同时完成写入正文及关闭连接。end 方法将会关闭 HTTP 连接。但因为我们同时还传入了一个字符串，end 方法将在把此内容发送给客户端后才关闭连接。

例子的最后一行调用了 console.log 方法。就像 Firebug 和 Web Inspector 支持的浏览器对应方法那样，它将在标准输出 stdout 上打印信息。

让我们在 Node.js 终端上运行此程序，并看看运行结果（例 1-10）。

例 1-10　运行“Hello World”程序

```
Enki:~ $ node
> var http = require('http');
> http.createServer(function (req, res) {
...     res.writeHead(200, {'Content-Type': 'text/plain'});
...     res.end('Hello World\n');
...   }).listen(8124, "127.0.0.1");
> console.log('Server running at http://127.0.0.1:8124/');
Server running at http://127.0.0.1:8124/
node>
```

在这里我们运行 Node 解析器，然后输入例子中的代码（我们不介意你从网站上进行复制粘贴）。Node 解析器接受了代码，并用“…”提示你的输入未完成并等待补充完整。当我们运行到 console.log 那行时，Node 解析器打印出 Server running at http://127.0.0.1:8124/。现在我们可以在浏览器中访问“Hello World”例子了（如图 1-1 所示）。

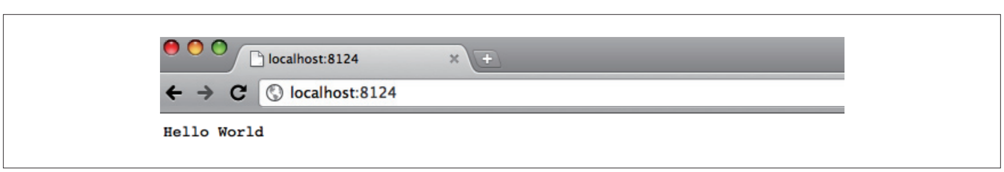


图 1-1：在浏览器中访问“Hello World”

跑起来了！虽然这不是一个惊人的演示，但我们只用了 6 行代码就把“Hello World”程序运行起来了。我们并不推荐这样的代码风格，但我们已经往前迈出第一步了。在下一章中，我们将看到更多的代码，但接下来我们先思考一下为什么 Node 会发展成为现在这个样子。

1.3 为什么选择Node

在写本书的时候，我们清楚地知道 Node.js 是多么地新。许多平台要经历多年才被人们接受，而在 Node.js 这一崭新的平台上，人们却展现了前所未有的热情。我们希望通过探究人们热衷于 Node.js 的原因，帮助你找到能产生共鸣的特性。通过了解 Node.js 的强项，我们能发掘出它最擅长的领域。本节将讨论是哪些因素造就了 Node.js，它为何能快速流行。

1.3.1 高性能Web服务器

当我们在 10 多年前第一次开始编写 Web 应用的时候，Web 还非常小。当然，期间我们经历了 .com 泡沫。但那时从事互联网行业的人数还是相当少，创建的网站也没现在这么火热。时至今日，有了先进的 Web 2.0 和随时随地可以上网的手机，这对我们这些开发人员提出了更多的要求。我们不但要提供更复杂、更多交互、更接近生活的功能，而且有越来越多的用户通过各种设备频繁使用这些功能，这是一个极大的挑战。硬件在持续改进，同时我们也需要提高软件开发水平来支持这些需求。如果只是单纯地采购更多的硬件来支撑新功能和新用户，就不那么划算了。

Node 给 Web 服务器程序开发领域引进了事件驱动编程，来尝试解决这一问题。实践证明，虽然 Node 不是第一个尝试此方法的平台，但它是目前为止最为成功的平台，而且我们认为它使用起来也是最容易的。后续章节会详细分析事件驱动编程，在这里我们先对其进行简短的介绍。想象一下，你现在需要连接到一台 Web 服务器上获取一个网页，这在正常的 DSL 连接速度下通常需要花费 100 毫秒左右。如果连接的是一台普通的 Web 服务器，它会在服务器上为你的请求创建一个新的程序运行实例。该程序自顶向下运行（按顺序运行所有的函数）来响应请求并生成网页返回给你。这意味着该服务器在请求被满足前需要一直占用固定大小的内存，其中包括了把数据返回给你所要等待的 100 多毫秒。Node 则不是采用此方式，而是在同一个程序内服务所有的用户。每当 Node 需要等待一些费时的操作，比如等待确认你已经收到返回的数据时（好让它标记此请求已经完成），它就继续处理下一个用户的请求去了。我们对细节描述得还是太多了，但这些特性意味着 Node 在内存处理上比传统服务器程序高效得多，也就是能够同时快速地服务更多的用户。这是个巨大的成就，人们也为此而热爱 Node。

1.3.2 专业的JavaScript

人们喜欢 Node 的另一个原因是 JavaScript。Brendan Eich 在 1995 年发明了 JavaScript 语言，这是一门在 Netscape 浏览器上使用的简单脚本语言。令人惊讶的是，自从 JavaScript 出现以来，它已经不止运用在浏览器上了。早先 Netscape 服务器程序就支持 JavaScript 作为一门服务器端脚本语言（称为 LiveScript）。虽然 JavaScript 当时并没有在服务器端得到广泛应用，却毫不妨碍它在快速发展的浏览器市场上大受欢迎。JavaScript 和微软的 VBScript 展开了激烈竞争，都想成为 Web 上的主流开发语言。很难说明为什么 JavaScript 最终胜出，也许是因为微软允许 JavaScript 运行在 IE 浏览器上[2]，也许是因为 JavaScript 语言本身优势明显，无法不脱颖而出，总之它完胜了。于是，在 2000 年初期，JavaScript 已经成为 Web 开发语言的代名词，不只是在浏览器开发 HTML 的第一选择，而且是唯一的选择。

这和 Node.js 又有什么关系呢？首先我们要记得当 AJAX 革命发生，并且 Web 风头正劲的时候（想想 Yahoo!、Amazon、Google 等是多么风光），AJAX 中“J”的唯一选择就是 JavaScript，完全没有其他替代品。这导致整个行业急需大量优秀的 JavaScript 程序员。Web 成为一个真正意义上的平台，并且附带着 JavaScript 是其开发语言，这就要求我们这些 JavaScript 程序员去提升自身能力。JavaScript 被视为程序员的第二或第三门编程语言，这本身就反映出人们对其重要性的重新认识。此时涌现出许多专家，他们的努力使 JavaScript 越来越为人们所接受。

这一运动的带头人物当属 Douglas Crockford。他关于 JavaScript 的文章和视频很受欢迎，帮助许多程序员发现了这门备受指责的语言中所隐藏的内在美。许多使用 JavaScript 的程序员为了处理 HTML 和 XML 文档，把主要精力花费在了浏览器对 W3C DOM API 的不同实现上。可悲的是，DOM 可能是 API 中最丑陋的，而且各款浏览器的实现又是那么地不一致和不完整。也难怪过去十年里许多程序员都没有把 JavaScript 认作一门“严肃”的语言。最近，Douglas 关于 JavaScript 好处（the good parts）的论述让人们认识到这门语言虽有弊病，但仍存在许多宝贵之处，因而带动了此语言的振兴。

在 2012 年的今天，有越来越多的 JavaScript 专家倡导 JavaScript 代码应当精心编写、高性能、易维护。Douglas Crockford、Dion Almaer、Peter Paul Koch（PPK）、John Resig、Alex Russell、Thomas Fuchs 等许多专家对此进行了研究、提议和加工，其中最主要的是提供了程序库，这些程序库让全世界成千上万的专业 JavaScript

---

译注 2：IE 浏览器并非真的支持 JavaScript 或 ECMAScript，它支持的变种叫 JScript。近年来，JScript 完整地支持了 ECMA- Script 3，以及 ECMAScript 5 的部分特性。同时，JScript 还和 Mozilla 的 JavaScript 一样实现了一些专有的扩展，并且增加了某些非 ECMAScript 标准的特性。

程序员能以追求卓越的精神去从事自己的行业。jQuery、YUI、Dojo、Prototype、Mootools、Sencha 等许多程序库部署在各个网站上供大量用户每日使用。在 JavaScript 不但被接受，还被广泛应用和拥护的环境下，这样的平台也许比 Web 本身更为宽广。当这么多的程序员了解了 JavaScript 时，这样的普及就成为了它的一个明显优势。

如果你在一屋子 Web 程序员中调查他们使用什么语言，会了解到 Java 和 PHP 是最流行的，Ruby 可能是目前次流行的（或者说至少和 Python 流行程度相当），Perl 则依然有许多追随者。但几乎可以肯定，任何从事 Web 开发的人都使用过 JavaScript。虽然后端语言与浏览器直接割裂开来，但编程这事儿都脱离不了部署这一环节。各种各样的浏览器及浏览器插件允许使用不同的语言，但这些语言都不足以成为 Web 开发的通用语言。现在我们面前有这样一个单一且通用的 Web 开发语言，我们怎样才能把它放到服务器上去呢？

1.3.3 浏览器之战 2.0

在互联网初期，我们就经历了恶名昭著的浏览器之战。Internet Explorer 和 Netscape 在 Web 功能上竞争激烈，他们分别在各自的浏览器上添加各种不兼容别人的编程特性，而且不支持其他浏览器所具备的功能。对于那些编写 Web 程序的开发者来说，这些都是苦闷的来源，因为这使得 Web 开发非常烦琐。Internet Explorer 或多或少成为了该轮竞争的胜者，变成了主流浏览器。几年之后，当微软在 IE6 上裹足不前的时候，出现了一个新的竞争者：从 Netscape 的旧成员中诞生的 Firefox。Firefox 让浏览器市场风云再起，WebKit（Safari）和 Chrome 紧跟其后。其中最有趣的还是在浏览器市场中新的竞争情况。

与浏览器之战的第一轮交锋不同，今天的浏览器主要争夺两个战场：一是坚持上一次浏览器战争后出现的标准，二是性能。随着网站越来越复杂，用户都想要最快的体验。这意味着，浏览器不但要很好地支持 Web 标准和允许开发者进行优化，还要尽力在自己内部进行优化。以 JavaScript 为核心模块的 Web 2.0 时代，AJAX 网站已经成为新的战场。

每个浏览器都有各自的 JavaScript 解析器：Firefox 的 Spider Monkey、Safari 的 Squirrel Fish Extreme、Opera 的 Karakan，最后还有 Chrome 带来的 V8。这些解析器不断追求更快的性能，也为 JavaScript 制造了创新的环境。为了让自己的浏览器突围而出，厂商们将尽最大能力让它运行得越来越快。

第 2 章 编写有趣的应用

过去几年，编程变化的趋势是让复杂的应用变得越来越容易开发。我们当然不能错过此等好事，而 Node 是专注于创建网络应用的，网络应用就需要许多 I/O（输入 / 输出）操作。让我们一起创建几个 I/O 应用，来看看使用 Node 有多么简单，并且还能轻松扩展规模。

2.1 创建一个聊天服务器

我们生活在一个实时的世界里，有什么比聊天更加实时吗？那就让我们先写一个基于 TCP 的聊天服务器吧，并且支持 Telnet 连接。这很容易，而且能够完全用 Node 来编写。

首先，我们需要在 Node 中包含 TCP 模块，并创建一个新的 TCP 服务器（例 2-1）。

例 2-1　创建新的 TCP 服务器

```
var net = require('net')

var chatServer = net.createServer()

chatServer.on('connection', function(client) {
  client.write('Hi!\n');
  client.write('Bye!\n');

  client.end()
})

chatServer.listen(9000)
```

代码第一行，我们加载了 net 模块。这个模块包含了 Node 需要的所有 TCP 功能。接着，我们调用 net.createServer() 方法来创建一个新的 TCP 服务器。有了这个服务器，我们需要用它做点儿事。这里调用 on() 方法来添加一个事件监听器。每当有新的客户端通过网络连接接入服务器，就会触发 connection 事件，事件监听器就会调用我们指定的函数。

连接事件在调用回调函数时，会传给我们新客户端所对应的 TCP socket 对象的引用。我们把此引用命名为 client。调用 client.write()，就能发送信息给该客户端。目前，我们只是简单地发送"Hi!"和"Bye!"，然后调用 client.end() 方法来关闭连接。就这么简单，我们的聊天服务器已经初露端倪了。最后，需要调用 listen() 函数，好让 Node 知道监听哪个端口。让我们马上测试一下吧。

我们可以使用 Telnet[1]（大多数操作系统都自带此程序）来连接新服务器进行测试。首先，调用 node 命令并带上文件名来启动服务器。然后，打开 Telnet 连接到 localhost 的 9000 端口（这是我们在 Node 程序中指定的端口）。见例 2-2。

例 2-2　用 Telnet 连接 Node TCP 服务器

```
Console Window 1h
----------------
Enki:~ $ node chat.js
Chat server started

Console Window 2
----------------
Last login: Tue Jun  7 20:35:14 on ttys000
Enki:~ $ telnet 127.0.0.1 9000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hi!
Bye!
Connection closed by foreign host.
Enki:~ $
```

到目前为止，我们创建了一个服务器，它能够接受客户端的连接，并且在断开连接前发送了一小段内容。但这还不能称为聊天服务器，我们再来添加几个功能吧。首先，需要能收到客户端发送的消息（例 2-3）。

例 2-3　监听所有的连接请求

```
var net = require('net')

var chatServer = net.createServer()
chatServer.on('connection', function(client) {
```

注 1：如果是在 Windows 上，我们推荐使用免费的 Putty 程序来作为一个 Telnet 客户端。

```
  client.write('Hi!\n');

  client.on('data', function(data) {
    console.log(data)
  })

})

chatServer.listen(9000)
```

这里添加了另外一个事件监听器，调用的是 client.on()。注意，我们是在 connection 回调函数的作用域中添加的这个事件监听器，这样就可以访问到连接事件所对应的 client 对象。新监听器关注的是 data 事件，每当 client 发送数据给服务器时，这一事件都会被触发。接着要删掉 client.end() 这一行。如果关闭了和客户端的连接，又如何获得新的数据呢？（当然，说“再见”那一行同样也删掉了。）现在，无论我们发任何数据给服务器，它都会在终端打印出来。让我们看看例 2-4。

例 2-4　从 Telnet 发送数据到服务器

```
Console 1
-------------

Enki:~ $ node chat.js
Chat server started
<Buffer 48 65 6c 6c 6f 2c 20 79 6f 75 72 73 65 6c 66 0d 0a>

Console 2
------------
Enki:~ $ telnet 127.0.0.1 9000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hi!
Hello, yourself
```

发生什么事情了？我们运行服务器并用 Telnet 连上了它。服务器发送“Hi!”，然后我们回应“Hello, yourself”。这个时候，Node 吐出了一堆你从来没有见过的看似无用的数据。原来，因为 JavaScript 无法很好地处理二进制数据，所以 Node 特地增加了一个 Buffer 库来帮助服务器。Node 并不知道 Telnet 发送的是什么类型的数据，所以在我们告诉它该用什么编码前，Node 只能保存原始的二进制格式。打印的字符信息实际是十六进制字节数据（详见 4.3.3 节）。每个字节对应着字符串“Hello, yourself”中的一个字母或字符。如果需要，可以调用 toString() 方法来把 Buffer 数据翻译为可读的字符串格式；不需要的话，也可以保持二进制格式，因为 TCP 和 Telnet 都能处理它。

现在我们能够接收客户端发送的消息了，接下来要做的事情是让它们互相发送消息。要完成此功能，需要让它们互相通信。之前我们采用的是 `client.write()` 方法，可惜它只能和一个客户端通信，而我们需要照顾到所有客户端。为此可以创建一个列表，然后把希望与之通信的客户端都添加进去。当一个新的客户端出现时，就把它添加到列表中，然后利用此列表实现客户端之间的通信（例 2-5）。

例 2-5　客户端之间的通信

```
var net = require('net')

var chatServer = net.createServer(),
    clientList = []

chatServer.on('connection', function(client) {
  client.write('Hi!\n');

  clientList.push(client)

  client.on('data', function(data) {
    for(var i=0;i<clientList.length;i+=1) {
      // 把数据发送给所有客户端
      clientList[i].write(data)
    }
  })

})

chatServer.listen(9000)
```

我们来看看例 2-6，现在可以连接多个客户端到服务器上，看看它们是如何互相发消息的。

例 2-6　客户端互相发消息

```
Console 1
------------

Enki:~ $ node chat.js

Console 2
------------

Enki:~ $ telnet 127.0.0.1 9000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hi!
Hello, yourself
Hello, yourself

Console 3
------------
```

```
Enki:~ $ telnet 127.0.0.1 9000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hi!
Hello, youtrself
```

这次，服务器没有记录它收到的任何消息，而是把列表中的每个客户端都轮询一遍，并把消息转发出去。值得注意的是，当终端 2 发送消息后，消息转发到了终端 3 的 Telnet 客户端上，同时也发回给终端 2 的 Telnet 客户端。这是因为我们在发送消息的时候，并没有检查发送者是谁，只是简单地把消息转发给所有客户端。而且 Telnet 客户端也无法区分哪些消息是自己发送的，哪些消息是别人发送的。我们需要改进一下。在例 2-7 中，我们创建一个函数，处理发送给所有客户端的消息并解决这些问题。

例 2-7　改进消息发送

```
var net = require('net')

var chatServer = net.createServer(),
    clientList = []

chatServer.on('connection', function(client) {
  client.name = client.remoteAddress + ':' + client.remotePort
  client.write('Hi ' + client.name + '!\n');

  clientList.push(client)

  client.on('data', function(data) {
    broadcast(data, client)
  })

})

function broadcast(message, client) {
  for(var i=0;i<clientList.length;i+=1) {
    if(client !== clientList[i]) {
      clientList[i].write(client.name + " says " + message)
    }
  }
}

chatServer.listen(9000)
```

首先，在 `connection` 事件监听器上为每个 `client` 对象增加 name 属性。为什么我们能为 `client` 对象添加属性呢？因为闭包绑定了每个 `client` 对象和相应的请求。于是，在闭包内就可以利用 `client.remoteAddress` 和 `client.remotePort` 来创建 `client` 的 name 属性，其中 `client.remoteAddress` 是客户端所在的 IP 地址，`client.remotePort` 是客户端接收从服务器返回数据的 TCP 端口。当不同

的客户端从同一个 IP 发起连接时，它们各自会有唯一的 remotePort。以后再向 client 发送消息时，我们就能用此唯一标识来找到它。

我们还把处理 data 的事件监听器代码抽离到了 broadcast 函数中。这样，通过调用 broadcast 函数就可以把消息发送给所有客户端。这一次，我们把发起消息（data）的 client 对象也传递进去，以便于把它从接收消息的客户端列表中排除掉。我们还把 client.name 加到了要发送的消息上，好让其他客户端清楚消息来源。现在看看这个改进后的服务器的运行效果（例 2-8）。

例 2-8　运行改进后的聊天服务器

```
Console 1
---------

Enki:~ $ node chat.js

Console 2
---------

Enki:~ $ telnet 127.0.0.1 9000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hi 127.0.0.1:56022!
Hello
127.0.0.1:56018 says Back atcha

Console 3
---------

Enki:~ $ telnet 127.0.0.1 9000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hi 127.0.0.1:56018!
127.0.0.1:56022 says Hello
Back atcha
```

现在它已经能够更加友好地提供服务了，虽然还不是很完美，但进步还是很明显的。需要注意，你自己运行例子的时候所看到的端口数字几乎肯定和例子中的数字不同。因为不同的操作系统对端口范围的限制不一样，并且端口的指定与你已经使用了哪些端口有关系，有点随机因素在里面。也许你已经发现了，我们的服务器有一个致命的缺陷。 如例 2-9 所示，如果其中一个客户端断开了，服务器就会出大问题。

例 2-9　断开一个客户端会导致服务器出错

```
Console 1
----------

Enki:~ $ node book-chat.js  ❶

net.js:392 ❷
    throw new Error('Socket is not writable');
          ^
Error: Socket is not writable
    at Socket._writeOut (net.js:392:11)
    at Socket.write (net.js:378:17)
    at broadcast (/Users/sh1mmer/book-chat.js:21:21)
    at Socket.<anonymous> (/Users/sh1mmer/book-chat.js:13:5)
    at Socket.emit (events.js:64:17)
    at Socket._onReadable (net.js:679:14)
    at IOWatcher.onReadable [as callback] (net.js:177:10)
Enki:~ $

Console 2
---------

Enki:~ $ telnet 127.0.0.1 9000 ❸
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hi 127.0.0.1:56910!
^]
telnet> quit ❹
Connection closed.
Enki:~ $

Console 3
---------

Enki:~ $ telnet 127.0.0.1 9000 ❺
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hi 127.0.0.1:56911!
You still there? ❻
Connection closed by foreign host. ❼
Enki:~ $
```

我们和之前一样先启动服务器 ❶，然后连接几个客户端 ❸❺，但是当终端 2 中的客户端断开连接时 ❹，麻烦就来了。如果终端 3 再发送消息 ❻，即调用 `broadcast()` 的时候，服务器会往一个已经断开的客户端写入数据 ❷。当终端 2 的客户端断开的时候 ❹，它对应的 socket 已经无法写入或读取了。而对已经关闭的 socket 进行 `write()` 操作时，Node 程序会抛出异常。这将导致其他所有客户端掉线 ❼。显然，这么脆弱的程序是不能作为服务器的。

这个问题应该从两方面来解决。首先必须保证在一个客户端断开的时候，要把它从客户端列表中移除，防止它再调用 `write()` 方法。V8 引擎也会把相应的 socket 对象作为垃圾回收，并释放相应的内存。其次，要采用更保险的方式调用 `write()` 方法。我们要确保 socket 从上次被写入到现在，没有发生任何阻碍我们调用 `write()` 方法的事情。好在用 Node 很容易做到这两点。第一点的详细做法见例 2-10。

例 2-10　把聊天服务器改造得更加健壮

```
chatServer.on('connection', function(client) {
  client.name = client.remoteAddress + ':' + client.remotePort
  client.write('Hi ' + client.name + '!\n');

  clientList.push(client)

  client.on('data', function(data) {
    broadcast(data, client)
  })

  client.on('end', function() {
    clientList.splice(clientList.indexOf(client), 1)
  })
})
```

我们先处理断开连接的客户端。当一个客户端断开时，要把它从客户端列表中移除。这可以利用 `end` 事件来完成。一个 socket 断开连接时会触发 `end` 事件，表示它要关闭。此时，调用 `Array.splice()` 将客户端从 `clientList` 列表中移除。`Array.indexOf()` 方法用于找到客户端在列表中的位置，然后 `splice()` 把它从列表中移除。在此之后，下一个客户端调用 `broadcast` 方法时，已经断开的客户端将不会再出现在列表中了。

此外，我们还能做得更加保险，如例 2-11 的代码所示。

例 2-11　检查 socket 的可写状态

```
function broadcast(message, client) {
  var cleanup = []
  for(var i=0;i<clientList.length;i+=1) {
    if(client !== clientList[i]) {

      if(clientList[i].writable) {
        clientList[i].write(client.name + " says " + message)
      } else {
        cleanup.push(clientList[i])
        clientList[i].destroy()
      }

    }
  }
  // 在写入循环中删除死节点，消除垃圾索引
```

```
    for(i=0;i<cleanup.length;i+=1) {
      clientList.splice(clientList.indexOf(cleanup[i]), 1)
    }
  }
```

调用 `broadcast` 函数的时候，检查一下 socket 是否可写，以确保不会因为任何一个不可写的 socket 导致异常。不仅如此，发现任何不可写的 socket 后，还要通过 `Socket.destroy()` 方法将其关闭并从 `clientList` 中移除。注意，遍历 `clientList` 的过程中并没有移除 socket，因为我们不想在遍历过程中出现任何未知的副作用。现在我们的服务器更加健壮了。在真正部署之前，还有一件事情要处理，那就是记录这些错误（例 2-12）。

例 2-12　记录错误

```
chatServer.on('connection', function(client) {
  client.name = client.remoteAddress + ':' + client.remotePort
  client.write('Hi ' + client.name + '!\n');
  console.log(client.name + ' joined')

  clientList.push(client)

  client.on('data', function(data) {
    broadcast(data, client)
  })

  client.on('end', function() {
    console.log(client.name + ' quit')
    clientList.splice(clientList.indexOf(client), 1)
  })

  client.on('error', function(e) {
    console.log(e)
  })
})
```

为 `client` 对象的 `error` 事件添加了 `console.log()` 调用后，可以确保客户端发生的任何错误都会被记录下来。而之前增加的代码，则能够确保在客户端抛出错误的时候，不会因为异常而导致服务器停止。

2.2　我们也来编写个Twitter

前一个例子展示了用 Node 编写一个实时应用有多么容易。当然，你有时候还要开发 Web 应用。让我们用 Node 来创建一个类似 Twitter 的 Web 应用。首先，我们需要安装 Express 模块（例 2-13）。这个针对 Node 的 Web 框架为现有的 `http` 服务器模块添加了更多的扩展（如 MVC），使开发 Web 应用更加简单。

例 2-13　安 h 装 Express 模块

```
Enki:~ $ npm install express
express@2.3.12 ./node_modules/express
├── mime@1.2.2
├── connect@1.5.1
└── qs@0.1.0
Enki:~ $
```

使用 Node 包管理程序（npm）可以很容易地安装 Express。一旦安装好了此框架，就能创建一个基本的 Web 应用（例 2-14）。这个程序和我们第 1 章所编写的例子很像。

> 你可以在第 6 章和第 7 章了解更多关于 npm 的知识。

例 2-14　使用 Express 的基本 Web 服务器

```
var express = require('express')

var app = express.createServer()

app.get('/', function(req, res) {
  res.send('Welcome to Node Twitter')
})

app.listen(8000)
```

这段代码和第 1 章中的 Web 服务器代码范例像极了，只是我们引入了 express 模块而不是 http 模块。Express 会在后台调用 http 模块，因为 Node 会自动解析依赖关系，所以我们不需要为此操心。和使用 http 或 net 模块类似，我们调用 Server() 来创建服务器，并调用 listen() 来监听指定端口。不需要用 Express 为请求事件指定监听器，而是可以通过 HTTP 匹配的方式来调用对应的方法。这个例子中，调用 get() 方法时，我们为匹配第一个参数所指定 URL 的 GET 请求指定了回调函数。Express 增加了两样 http 模块所没有的功能：根据 HTTP 请求的不同方法进行过滤，根据特定的 URL 进行过滤。

至于回调函数，它看起来和 http 模块的方法很像，实际上就是一样的。此外，Express 还添加了若干其他方法。采用 http 模块时，我们需要创建 HTTP 头，并且在发送请求内容之前把 HTTP 头先发送给客户端。Express 提供了一个方便的方法，这就是 res(http.response) 对象的 send() 方法。此方法发送 HTTP 头，同时还会调用 response.end() 方法。到此为止，我们这个例子比第 1 章的 Hello World 服务器例子强不了多少。只不过这个新的服务器只响应 URL 为“/”的 GET 请求，而且不会抛出错误，第 1 章的例子则会响应任何 URL 和任何请求。

下面让我们开始为服务器添加一些类似 Twitter 的功能吧（例 2-15）。在刚开始的时候，我们先不去管健壮性或者扩展性。假设不用考虑这些问题，好让你看清楚如何编写一个应用。

例 2-15　添加基础 API

```
var express = require('express')

var app = express.createServer()
app.listen(8000)

var tweets = []

app.get('/', function(req, res) {
  res.send('Welcome to Node Twitter')
})

app.post('/send', express.bodyParser(), function(req, res) {
  if (req.body && req.body.tweet) {
    tweets.push(req.body.tweet)
    res.send({status:"ok", message:"Tweet received"})
  } else {
    //没有 tweet？
    res.send({status:"nok", message:"No tweet received"})
  }
})

app.get('/tweets', function(req,res) {
  res.send(tweets)
})
```

在前面简单的 Express 应用基础上，我们添加了几个函数来提供最基础的 API。但先看一下我们做的另外一个修改。我们把 `app.listen()` 调用移到了文件的上方。为什么调用放到前头不会导致下面响应请求的函数出问题呢？理解这一点非常重要。你可能会认为，如果在文件开头调用 `app.listen()`，那么从调用 `app.listen()` 结束到解析完下面的函数这段时间里，所有到来的请求都将得不到处理。这样的想法并不正确，原因有二。第一，JavaScript 所有的事件触发都是在事件循环中，这意味着除非我们已经完成了这次循环中的所有处理函数，否则新的事件是不会被触发调用的。对这个例子而言，除非我们已经把文件中所有的初始化代码执行完，否则不会调用 `request` 事件（从而也就不会调用相应的处理函数）。第二，`app.listen()` 函数调用是异步的，因为绑定 TCP 端口也需要花时间，而其他（通过 `app.get()` 和 `app.post()` 指定的）事件监听器则是同步的。

我们通过 `app.post()` 方法添加“/send”的 POST 路由来提供基础的 tweet 功能。这个函数对比之前的例子有些难懂。显然，这是一个 `app.post()` 而不是 `app.get()` 方法，这就意味着它接受的是 HTTP POST 请求而不是 HTTP GET 请求。更

加显著的不同点是，我们为此函数多传入了一个参数。其实，你不需要对所有的 app.post() 调用都做这个操作，甚至可以全都不传此参数。这个跟在 url 后面的参数是个中间件。

中间件是指一小段特定的代码，位于原始请求事件与我们给 app.post() 指定的路由之间。我们通过中间件对一些通用功能进行代码重用，如用户授权或者 log 记录。在本例中，中间件的作用是把客户端 POST 的数据转换成我们能够使用的 JavaScript 对象。这是 Express 本身就自带的 bodyParser 模块。我们在指定 app.post() 路由的时候把它包含进来就可以了，就是调用 express.bodyParser()。这个函数调用会返回另外一个函数。这种调用中间件的标准方式可以让你在需要的时候指定中间件的配置。

如果我们不使用中间件，就需要自己动手写代码处理请求对象（res）所提供的数据了。只有当 POST 传输的所有数据都接收完了，才会调用 app.post() 指定的函数。使用中间件不但能将代码重用，还使代码结构更加清晰。

express.bodyParser 为 req 对象添加了新的属性，称为 req.body。这个属性（如果它存在的话）包含了 POST 数据对应的对象。express.bodyParser 中间件只能够处理 POST 方法的数据，而且要求 HTTP 头的 content-type 属性是 application/x-www-form-urlencoded 或 application/json。这两种数据格式都能够很容易地解析成 key/value 值，并保存到 req.body 对象中。

在 app.post() 处理函数中，我们第一步要先检查 express.bodyParser 是否找到了数据，只要检查一下 req.body 是否存在就可以了。如果存在，我们查找 req.body.tweet 的这一属性，这是 tweet 的内容。如果找到了 tweet，就把它记录在一个叫做 tweets 的全局数组中，并且返回客户端一个 JSON 字符串表示已经成功。如果没有找到 req.body 或 req.body.tweet，就返回表示失败的 JSON 字串给客户端。注意，我们并没有在调用 res.send() 时对数据进行序列化。如果传给 res.send() 一个对象，它会自动把其序列化为 JSON 并添加对应的 HTTP 头。

最后，我们把基础的 API 补充完整，添加了一个监听"/tweets"的 app.get() 路由。这个路由只是简单地把 tweets 数组的内容以 JSON 的形式返回出去。

我们可以写一些测试来确认这些简单的 API 能正常工作（例 2-16）。即使你不是采用完整的测试驱动开发（TDD），这个做法也是一个良好的习惯。

例 2-16　测试 POST API

```
var http = require('http'),
    assert = require('assert')
```

```
var opts = {
  host: 'localhost',
  port: 8000,
  path: '/send',
  method: 'POST',
  headers: {'content-type':'application/x-www-form-urlencoded'}
}

var req = http.request(opts, function(res) {
  res.setEncoding('utf8')

  var data = ""
  res.on('data', function(d) {
    data += d
  })

  res.on('end', function() {
    assert.strictEqual(data, '{"status":"ok","message":"Tweet eceived"}')
  })
})

req.write('tweet=test')
req.end()
```

我们需要 `http` 模块来发送 HTTP 请求，然后用 `assert` 模块 [2] 对返回值进行测试。`assert` 是 Node 的一个核心模块，它能帮助我们用多种方式来对返回值进行测试。当一个值与预期的条件不符时，将抛出异常。通过测试脚本来检查程序运行时应该有的表现，可以确保它的功能正确。

`http` 模块并非只包含了 HTTP 服务端的功能，它同时还提供了客户端的功能。在这个测试程序中，我们使用 `http.request()` 这一工厂方法来创建新的 `http` 请求对象，并指定了 `options` 这个参数。我们通过配置 `options` 的一系列属性，来让 `http.Request` 对象按我们的要求运行。在创建其他 Node 对象的时候，你也会看到类似的配置方法。在上面的例子中，我们指定了主机名（会被 `dns` 解析）、端口、URL 路径、HTTP 方法和一些 HTTP 头。这些信息都是我们创建 Express 服务器所采用的。

`http.request()` 构造函数接受了两个参数：第一个是 `config` 对象，第二个是回调函数。回调函数是监听在 `http.Request` 的 `response` 事件上的。这与 `http.Server` 很类似，但这里返回对象只会出现一次。

我们处理返回信息的第一步是调用 `setEncoding()` 方法。这就让我们指定了所有接受数据的编码方式。通过设置为 `utf8`，我们确保接收的数据都被正确地处理为需要的字符串格式。下一步我们定义了一个变量 `data`，用它以流式方式处理来自服务器的所有响应数据。在 Express 服务器中，我们用了 `express.bodyDecoder` 来处

注 2：你可以在第 5 章获得更多关于 assert 模块的信息。

理请求数据流中所有的内容，但在客户端我们没有这么高级的工具，所以需要手动处理流数据。其实这也非常简单，只需指定响应返回时的 `data` 事件监听函数。当一部分数据到达时，把它追加到 `data` 变量上就行了。同时，我们监听响应的 `end` 事件，以便在所有数据都到达后采取行动。API 之所以设计成这种方式，是因为许多应用可以利用它来进行数据流的操作，也就是可以一边接收数据一边处理，而不需要先等待所有数据都发送完成再操作。

只有当我们把服务器返回的数据接收完整后，`end` 事件才会被触发。现在我们可以对服务器返回的数据进行测试了。测试用例中将检查 `data` 变量中的数据是否和我们预期服务器会发送的内容一致。如果服务器运行正常，它将返回一个 JSON 数据。利用 `assert.strictEqual` 函数，我们能对数据进行“===”级别的一致性检查。如果检查条件不满足要求，就会抛出异常。因为要模拟网页表单的操作，所以需要采用 `x-www-form-urlencoded` 格式发送数据。

现在我们准备好了请求对象和相关的事件处理函数，下一步是往服务器发送数据。我们调用 `write()` 函数来发送数据（因为这是一个 POST 请求），通过发送一些测试数据来检查服务器的响应内容是否正确。最后，调用 `end()` 方法来表示数据已经发送完毕。

运行这个脚本，它将连接到我们启动好的服务器（如果它正在运行）并发送 POST 请求。如果它接收到了正确的数据，就会静静地退出。反之，如果它连接不上服务器，或者服务器返回的内容不是预期的数据，脚本将抛出异常。编写这些测试脚本的目的，就是检查服务器是否达到了设计要求。

有了这个 API 以后，我们就能进一步开发 Web 界面来供用户使用了。现在功能还很基础，但 API 已经能让用户向所有人发送消息了。让我们来写个界面吧。

Express 围绕着请求路由的方式支持 MVC 结构（模型，视图，控制器）。控制路由与控制器类似，提供了把数据模型和视图相结合的方法。我们已经使用过路由（`app.get('/', function)`）。在例 2-17 的文件夹结构中，存放了视图的不同部分。通常，views 文件夹存放的是视图模版，在其中的 partials 文件夹中包含的是子模版（在后面会详细介绍）。对于不使用内容分发网络（CDN）的应用，public 文件夹存放的是静态文件，如 CSS 和 JavaScript。

例 2-17　Express 应用的基本文件夹结构

```
.
├── app.js
├── public
└── views
    └── partials
```

要把我们简单的模型（`var tweets = []`）连接到视图中，需要先创建一些视图。首先在 views 文件夹下创建视图文件。Express 提供了几种不同的模版语言，并且可以为其扩展更多类型。我们首先采用 EJS 模版语言[3]。EJS 是通过把 JavaScript 嵌入在模版中，并提供了一些简单的标签来定义 JavaScript 该如何解析运行。我们来看一个 EJS 的例子，先由例 2-18 中的 layout 文件开始。

例 2-18　EJS 布局模板文件

```
<!DOCTYPE html>
<html lang="en">
    <head>
      <meta charset="utf-8">
      <%- partial('partials/stylesheet', stylesheets) %>
      <title><%= title %></title>
    </head>
    <body>
        <h1><%= header %></h1>
        <%- body %>
    </body>
</html>
```

Express 的 layout 文件定义了网站的骨架。它是你在几乎所有地方都需要使用的基本视图样式。在本例中，我们采用了一个非常简单的 HTML5 页面。页面头部包含了一些样式表定义，然后是正文内容。正文包含了一个 `h1` 头元素和其他一些内容。注意 `<%` 标签，这是我们将插入 JavaScript 变量的地方。在 `<%` 和 `%>` 标签之间的 JavaScript 会被执行。我们后面会再详细介绍这些以 `=` 或 `-` 开头的标签。通常，你只需要引用一些数据（如变量或引用）放到页面中来，比如，`<h1><%= header %></h1>` 将 `header` 变量包含在了 `h1` 元素中。

模版中有两个特殊的地方。第一个是 `partial()`，Partial 是一些迷你模版，可以通过指定不同的数据重复使用。例如，你可以想象一篇博客的评论内容，它就是一段固定的 HTML 格式重复了许多遍，只是其中的评论者和评论内容不一样。Partial 是保存可重复利用的代码片段的地方，它独立于使用它的页面，这样在更改时只需要修改一个地方便能同时改变所有页面的样式。另外一个特殊的用法是 `body` 变量。因为我们在网站的所有页面都用了这个 layout 格式（除非主动关闭它），所以需要有一些方法来指定不同页面所需要渲染的内容。Express 为此提供了 `body` 变量，这个变量包含了我们想要渲染的模版。

在继续查看其他模版之前，我们先指定一个路由来调用 `render` 函数生成页面看看效果（例 2-19）。

注 3：第 7 章会介绍更多 Express 的模版语言。

例 2-19　为 '/' 路由渲染 index 模板

```
app.get('/', function(req, res) {
  var title = 'Chirpie',
      header = 'Welcome to Chirpie'

  res.render('index', {
    locals: {
      'title': title,
      'header': header,
      'tweets': tweets,
      stylesheets: ['/public/style.css']
    }
  })
})
```

这个路由处理的代码和我们之前使用的其他路由代码类似，只不过这次不是调用 res.send()，而是调用 res.render() 函数来渲染一个模版。第一个参数是我们想要渲染的模版的名字。需要记住，无论 index 模版渲染成什么样子，它都会放入 layout 模版中 body 变量所在的位置。传给 res.render() 的第二个参数是配置对象，这里我们并未作任何配置，只指定了一些本地变量。配置对象中的 locals 属性包含了需要渲染此模版的数据。我们指定了 title、header、tweets 数组和 stylesheets 数组，所有这些变量都在 layout 和 index 模版中使用了。

我们需要在 index 模版中定义渲染 tweet 的方法，好让用户能够看见所有提交的消息（例 2-20）。我们不会单独显示每条 tweet 流数据，而是会提供一个页面，让每个人都能看到所有提交的内容，并可以通过 API 提交各自的消息。

例 2-20　展示 tweet 和让用户提交新 tweet 的 index 模板

```
<form action="/send" method="POST">
  <input type="text" length="140" name="tweet">
  <input type="submit" value="Tweet">
</form>
<%- partial('partials/chirp', tweets) %>
```

这个 index 模版非常简单。我们提供了一个短的表单来输入新的 tweet 信息。这是普通的 HTML 方法，之后可以改为使用 AJAX 方法。接着是渲染 tweet 用的 partial 模版。因为它们都是一样格式的，我们不想在 index 模板里使用丑陋的循环把标记嵌入进去。通过使用 partial 模版，我们把呈现 tweet 的简短模版抽象出来，在需要的地方重复利用。这样做能够保持代码的美观和简洁。我们还能在之后增加更多的功能，但现在已经提供了所需要的基本功能了。我们还需要定义在 layout 和 index 模版中用到的 partial 模版（参见例 2-21 和例 2-22）。

例 2-21　渲染 chirp 的 partial 模版

```
<p><%= chirp %></p>
```

例 2-22　渲染 stylesheet 的 partial 模版

```
<link rel="stylesheet" type="text/css" href="<%- stylesheet %>">
```

这两个模版都超级简单，他们接收一些输入数据然后插入指定的地方。因为传入的参数是一个数组，所以模版渲染时会自动遍历数组中的每一个元素。这里并没有什么高深的做法。每个子模版接收的数组变量名称和模版的名称一致，如名叫 `chirp` 的模版访问的数据变量名称也叫 `chirp`。例子中的数据都是简单的字符串，如果我们传入的是一组对象，可以通过 `chirp.property` 或者 `chirp['property']` 的方法来获得这些对象对应的 propery 名称的属性。当然，你还能调用方法，如 `chirp.method()`。

现在，我们的应用允许用户提交 tweet 了。它很简单，但还有一些地方可以改进。让我们现在就改正这些问题吧。第一个显然的问题是，当我们提交新的 tweet 时，会进入到发生 JSON 对应的处理代码。虽然我们访问 URL 的路径为 `/send` 没什么大问题，但对服务器来说不应该对所有用户一视同仁。而且 tweet 只是按时间顺序加入，缺少了时间戳，我们无法知道它们的新旧程度。我们也需要解决这个问题。

处理 `/send` 路径很简单。当 HTTP 客户端发送请求时，它可以指定想要返回的数据格式。通常浏览器会优先请求 `text/html` 格式，再考虑其他格式。然而，在调用 API 请求时，客户端可能会指定 `application/json` 格式来获得需要的输出内容。通过检查 HTTP 头的 `accept` 字段，我们可以确定对于浏览器返回的是主页，而 API 客户端收到的是 JSON。

HTTP 头的 `accept` 字段可能是 `text/html,application/xhtml+xml,application/xml;q=0.9,*/*; q=0.8`。这是从 Chrome 浏览器发送上来的，包含了一连串以逗号分隔的 MIME 类型。我们首先需要一个小方法来确定 `text/html` 是否在 `accept` 字段中（例 2-23），然后根据测试的结果执行不同的逻辑。

例 2-23　检查 `accept` 头是否包含 `text/html` 的小函数

```
function acceptsHtml(header) {
  var accepts = header.split(',')
  for(i=0;i<accepts.length;i+=0) {
    if (accepts[i] === 'text/html') { return true }
  }

  return false
}
```

这个函数把头字段根据逗号切割成数组，然后遍历这些字段，看其中是否有字段与 `text/html` 匹配，有则返回 `true`，否则返回 `false`。我们通过这个函数来区分请求来源是浏览器还是 API（例 2-24）。

例 2-24　重定向浏览器到 /send

```
app.post('/send', express.bodyParser(), function(req, res) {
  if (req.body && req.body.tweet) {

    tweets.push(req.body.tweet)

    if(acceptsHtml(req.headers['accept'])) {
      res.redirect('/', 302)
    } else {
      res.send({status:"ok", message:"Tweet received"})
    }

  } else {
    // 没有 tweet ?
    res.send({status:"nok", message:"No tweet received"})
  }
})
```

大部分代码和例 2-10 中相同，但增加了头信息中 accept 字段是否包含 text/html 的检查。如果包含，我们就调用 res.redirect 指令告诉浏览器重定向到 / 去。需要返回 302 状态码，因为这不是永久性地改变路径，我们想让浏览器每次发送 tweet 时还是以 /send 为入口。

第3章

编写健壮的Node程序

要想利用好服务端的 JavaScript 环境，关键是要理解 Node.js 和 JavaScript 设计决策背后的一些核心理念。理解设计决策及其利弊得失能帮助你更轻松地写出好代码并做好系统架构，还能帮助你向别人解释清楚，为什么 Node.js 和他们平时用的系统不一样，其性能是如何得到提高的。工程师都不喜欢自己的系统中有不清楚的地方，他们不接受用“魔力”来笼统地解释一切，而需要清楚地知道一个特定的架构为何能带来益处，以及在什么样的情景下能带来益处。

本章将涵盖代码风格、设计模式、产品诀窍等，而这些都是写出既优秀又健壮的 Node 代码所需要了解的。

3.1 事件循环

Node 的一个核心功能就是*事件循环*，这一概念也多用于 JavaScript 底层行为及许多交互系统中。在许多语言中，事件模型是在外层的，但 JavaScript 事件一直是其语言的核心模块，这是因为 JavaScript 在很多情景下都需要处理与用户交互的事件。用过现代网页浏览器的人都习惯在网页上通过 `onclick`、`onmouseover` 等事件来进行操作。这些事件是那么常见，我们在开发网页交互的时候甚至会忘记它们的存在，但在语言内部支持事件模型是何等强大的功能！在服务器端，没有了网页 DOM 对应的那些有限的用户驱动型交互事件，而是在服务器程序上对应发生的各种不同的事件，比如 HTTP 服务器模块在用户发送请求给 Web 服务器时会触发 `request` 事件。

JavaScript 利用事件循环来合理地处理系统各部分的请求。在计算领域，人们可以

用若干不同的方法来处理实时或并行运算，但大多数方法都太过复杂，甚至让人头疼。JavaScript 采用了很简易的方法，使得处理过程更容易理解，但是它需要有一些限制条件。当你把握了事件循环的工作原理后，就能充分地扬长避短了。

Node 采用的方式是，所有的 I/O 事件都应该是非阻塞的（稍后会解释原因）。这意味着需要让程序暂停操作的 HTTP 请求、数据库查询、文件读写，以及其他事情在数据返回之前并不暂停执行。这些事件都将独立运行，然后在数据准备好以后触发一个事件。也就是说，用 Node.js 编程会用到很多回调函数，来处理各种 I/O。回调函数往往以级联的方式嵌在其他回调函数中，这与浏览器编程有所不同。除了用顺序的方式设置好启动项外，大部分代码都是在处理回调函数。

针对这种少见的编程风格，我们需要寻找适合服务器编程的处理模式。先从事件循环开始吧。我们认为大部分人直觉上是理解事件驱动编程的，因为这和日常生活很像。假设你在烧饭，正在切青椒的时候锅里的东西开始沸溢了（图 3-1），你会暂停切菜，把炉火关小。你不会在切青椒的同时把炉火关小，而是会采用更加安全的方式，通过快速切换工作对象来达到同样的目的。事件驱动编程也是同样的道理。通过让程序员一次只能为一个回调函数编写处理代码，可以让代码可读性更强，而且能够快速地处理多个任务。

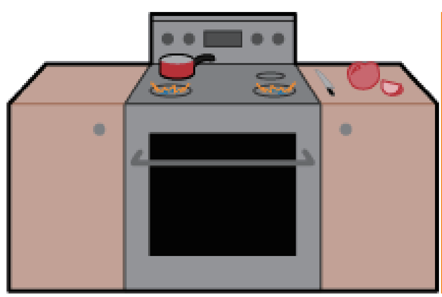


图 3-1：事件驱动的人们

在日常生活中，我们习惯于用各种内部回调的方式来处理遇到的事件。和 JavaScript 类似，我们一次只能处理一件事情。好吧，我知道你可以同时揉肚子和拍脑袋，并且能两样都干得不错，但当你想同时做一些重要的事情时，很快就会出差错的。这点也和 JavaScript 很像，能让事件来驱动操作很棒，但它只能以“单线程”的方式运行，即同一时间只能处理一件事情。

单线程的概念非常重要。常有人批评 Node.js 缺少并发，也就是它没有利用机器上的所有 CPU 来运行 JavaScript。但是，同时在多个 CPU 上运行程序也有它的问题，

是需要协调多个执行线程的。要让多个 CPU 有效地拆分任务，它们之间需要不停地交换信息，比如当前执行状态，以及各自完成了哪些工作。虽然这不是不可能，但这么复杂的模型给程序员和系统带来了很大的工作量。JavaScript 的方式很简单：同一时刻只有一件事情在操作。Node 做的每一件事情都是非阻塞的，所以事件触发与 Node 对其操作的时间间隔是很短的，因为 Node 不需要等待如磁盘 I/O 这样的操作。

再举个邮递员投递的例子，帮助你理解事件循环。邮递员的每封信都是一个事件，他有一堆事件等着要按顺序处理，每封信（事件）都要走到相应的路径进行投递（图 3-2）。路径就是对此事件的回调函数（通常不止一条路径）。可怜的是，我们的邮递员只有一双腿，每次只能走其中一条路径。

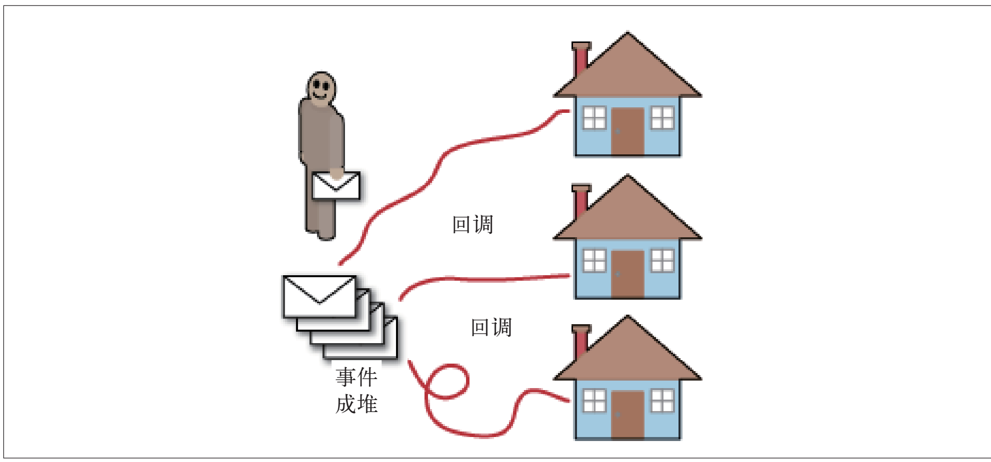


图 3-2：事件驱动的邮递员

偶尔，当邮递员在路上行走时，有人会给他另外一封信件，这就像是投递途中的回调函数。这种情况下，邮递员会马上去派送新的信件（因为路人不去邮局而是直接交给他的信件，一定是十万火急的）。此时邮递员会立刻切换到新的路径去投递新邮件，完成后，再回到之前的路径上继续工作。

让我们从简单情形入手，对比一下邮递员的行为和一般程序的做法。假设我们的 Web 服务器（HTTP）被请求要从数据库中读取一些数据，然后返回给用户。在这种情况下，我们只要处理很少的事件。首先，用户的请求多是要 Web 服务器返回一个网页。处理这个初始请求的回调函数（我们称之为回调函数 A）会先从请求的对象中确定它要从数据库读取什么内容，然后向数据库发起具体的请求，并传入一个函数（回调函数 B）供请求完成时使用。处理完请求后，回调函数 A 结束并返回。当数据库找到需要的内容后，再触发相应事件。事件循环队列则调用回调函数 B，让它把数据发送给用户。

这似乎非常直观。这里需要特别注意的是代码“隔断”的地方，这也是过程式的程序不会遇到的情况。因为 Node.js 是一个非阻塞的系统，所以当调用需要阻塞等待的数据库函数时，我们会采用回调函数替代闲置等待。这就是说，由另外一些函数来接管这个请求，并在数据准备好返回时把它处理掉。所以我们需要确认回调函数所要用到的数据能够有办法取得。JavaScript 编程通常是利用闭包来实现这个功能的。稍后，我们会进一步介绍闭包。

为什么 Node 更加高效呢？想象一下在一家快餐店点餐。你在柜台排队时，服务员有两种方法来处理你的点单，一种是事件驱动的，另一种则不是。我们先采用 PHP 等许多 Web 平台所使用的方法。你点餐时，服务员先招待你，待你点完后才服务下一个客人。他输入完你的单子后，可以做以下几件事情：收款、为你倒饮料等。但是，服务员还不知道要等多久厨房才能够把你要的汉堡做好（如果你们中有一人是素食主义者，可能还要等更长时间）。在传统的 Web 服务框架下，每个服务程序（线程）每次只能服务一个请求。唯一增加处理能力的方法就是加入更多的线程。很显然这样的做法并不是那么地高效，服务员在等待厨房做菜时浪费了很多时间。

显然，现实生活中的餐馆使用的是更加高效的模式。你点完菜后，服务员会给你一个号码，在菜做好时通知你，你可以称这个为回调号码。Node 也是这样工作的。当 I/O 一类的费时操作开始时，Node 会给它们一个回调引用，然后继续处理其他已经就绪的工作。比如说服务员可以服务下一个客人（对 Node 来说，则是下一个事件）。需要重点关注的是，与邮递员的例子一样，餐厅服务员也绝不会在同一时间服务两个客人。当呼叫某位客人来取食物的时候，他们不会处理新客人的需求，反之也是一样。通过事件驱动的运作方式，服务员能够最大程度地提高产出。

下面这个例子展示了在什么样的情况下使用 Node 最合适，以及什么情况下它不合适。在一些小餐馆，厨师和服务员是同一个人，这种情况下采用事件驱动并不能提高效率，因为所有的工作都由同一个人完成，事件驱动的架构并不能增加价值。如果服务器的全部（或大部分）工作是进行运算，Node 并非最理想的模型。

同时，我们也能发现这个架构在什么时候合适。假设在餐馆中有两名服务员和四位客人（图 3-3）。如果服务员一次只服务一位客人，那么头两位客人可以最快地拿到食物，而第三和第四位客人的体验会很糟糕。前两位客人之所以能够快速地获得食品，是因为服务员在全力满足他们的要求，这占用了另外两位客人的时间。在事件驱动模型下，头两位客人可能需要稍微等待一下才能拿到食物，因为服务员需要先处理一下后面两位客人的点单，但系统的平均等待时间（延迟）将大大降低。

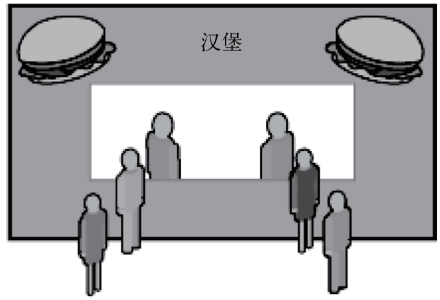


图 3-3：快餐，快捷代码

现在我们看看另外一个例子。我们给事件循环模式的邮递员一封信去投递，但投递这封信需要经过一扇门。他到达了目的地，而门却关闭着，所以他只能等待并不停地尝试进入。他等待门打开就像进入了死循环模式（图 3-4）。如果在信件队列里有另外一封信能够通知某人来打开门，让邮递员进去，这不就解决问题了吗？不幸的是，邮递员正在无休止地等待打开门，无法抽身去投递那封信，这是因为打开门的事件是在当前回调事件的外部。如果在回调函数内发起事件，我们知道邮递员会优先把这封信给投递掉，但是当事件是在当前执行代码的外部发生时，它必须等待正在执行的代码完成之后才会被调用。

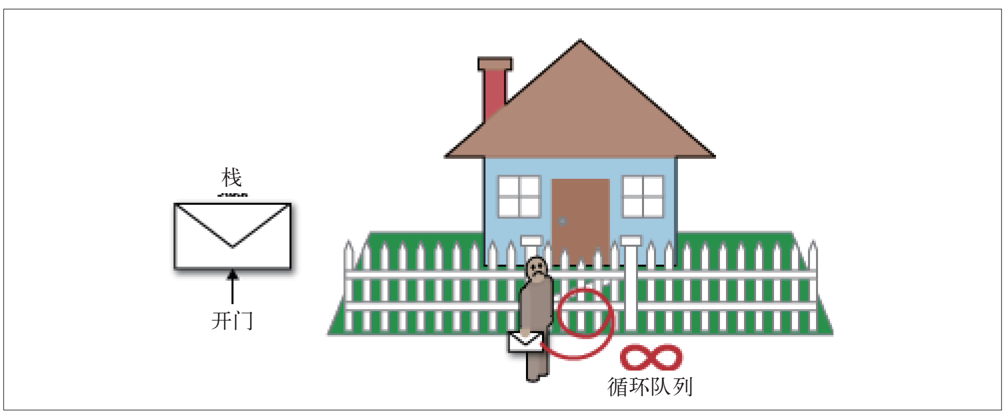


图 3-4：被堵塞的事件队列

如例 3-1 所示，Node.js（或浏览器）创建的事件永远不会跳出。

例 3-1　堵塞事件循环的代码

```
EE = require('events').EventEmitter;
ee = new EE();
```

```
die = false;

ee.on('die', function() {
    die = true;
});

setTimeout(function() {
    ee.emit('die');
}, 100);

while(!die) {
}

console.log('done');
```

在上面的例子中，`console.log` 永远不会被调用，因为 `while` 循环不会让 Node 有机会触发 `timeout` 回调函数并且发起 `die` 事件。虽然我们不太会写这样一个依赖外部条件作为跳出判断的循环体，但它展示了 Node.js 同时只处理一件事的本质，任何一点缺陷都可能导致整个系统混乱。这也是事件驱动编程的核心模块是非阻塞 I/O 的原因。

我们再做一下算术。当 CPU 进行一次运算的时候（不是一行 JavaScript 代码，而是单一的机器码运算）大概需要 1/3 纳秒（ns）。一个 3Ghz 的处理器每秒运行 3 × 109 个指令，所以每个指令花费 10-9/3 秒。主流的 CPU 内部有两种内存，L1 和 L2 cache，访问速度大概为 2~5 纳秒。如果我们从内存（RAM）读取数，需要花费大概 80 纳秒，比运行指令要慢两个数量级。但这些操作都是在同一个场景下的。从更慢的 I/O 途径中读取内容则太糟糕了。如果把从 RAM 中读取数据比作一只猫的重量，那么从硬盘上读数据就比得上一头鲸了，而从网络上等数据就像是 100 头鲸的重量。假设拿运行 `var foo = "bar"` 与一个数据库查询对比，简直就是一只猫和 100 头鲸比重量。阻塞式 I/O 并非真的在事件循环的邮递员前面放了一扇真实的门，它只是把邮递员送到遥远的非洲大陆后再回来投递信件。

有了事件循环的基本认识后，我们看看常用的 Node.js 代码是如何创建 HTTP 服务器的（例 3-2）。

例 3-2　基本的 HTTP 服务器

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(8124, "127.0.0.1");
console.log('Server running at http://127.0.0.1:8124/');
```

这是 Node.js 网站上展示的最简单例子（但稍后我们会说明这并非编码的理想方式）。在例子里，通过调用 `http` 库的一个工厂方法来创建 HTTP 服务器。工厂方法

在创建新的 HTTP 服务器的同时，为 `request` 事件绑定了一个回调函数，后者作为 `createServer` 的一个参数传递进去。当代码运行的时候会发生什么有趣的事情呢？ Node.js 运行的第一件事情是把例子中的代码从头到尾运行一遍，这可以认为是 Node 编程的“设置”阶段。因为我们绑定了一些事件监听器，所以 Node.js 不会退出，而是等待这些事件被触发。如果我们没有绑定任何事件，Node.js 在运行完代码后就会立刻退出。

那么当服务器接收到一个 HTTP 请求时会进行什么处理呢？ Node.js 会发起 `request` 事件，因为该事件有对应的回调函数绑定在上面，回调函数会被依次调用。在本例中，只有一个回调函数，那就是在调用 `createServer` 时作为参数传入的匿名函数。我们假设在服务器启动以后来了第一个请求，因为这个时候没有任何其他代码在运行，所以这个 `request` 事件被马上处理并调用回调函数。这是个极为简单的回调过程，所以运行飞快。

假设我们的网站变得非常受欢迎，同时有很多的请求进来了。为了方便讨论，假设回调函数需要执行 1 秒钟。在第一个请求后紧跟着又来了第二个请求，那么第二个请求将不会在这 1 秒内被处理。显然 1 秒钟其实是很长的时间了。让我们看看真实应用情景，事件循环堵塞的问题会严重地破坏用户体验。HTTP 服务器实际上是由操作系统内核处理与客户端的 TCP 连接的，所以尽管不会恶化到拒绝新连接的境地，但仍然会有这些链接不被处理的危险。为了处理这些问题，我们希望尽量保持 Node.js 的事件驱动和非阻塞的特性。同样的方式，让费时的 I/O 事件用回调的方法来通知 Node.js，只有数据已经准备好了，才可以进行下一步操作。Node.js 程序本身需要把每一个回调函数都写得运行迅速，防止把事件循环给堵塞住。

这意味着你在编写 Node.js 服务器程序的时候需要遵循以下两个策略。

- 在设置完成以后，所有的操作都是事件驱动的。
- 如果 Node.js 需要长时间处理数据，就需要考虑把它分配给 web worker 去处理。

事件驱动方法配合事件循环工作起来非常高效（正如它的名字所暗示的），但编写容易阅读和理解的事件驱动代码也同样重要。在前面的例子里，我们用匿名函数作为事件回调，这会导致几点不便。首先，我们无法控制代码在哪里使用。匿名函数只有在被使用的地方才存活，而不是在绑定事件回调时存活，这会影响调试。如果所有东西都是匿名事件，当异常发生时，就很难分辨出是哪个回调函数导致了问题。

3.2 模式

事件驱动编程和过程式编程是不一样的，最简单的学习方式是参照前人实践过的常

用模式。这正是本节的目的。

在介绍模式之前，我们先看一下各种编程风格背后发生了哪些事情，以便确定讨论模式的语境。本节主要探讨 I/O 问题，正如 3.1 节所讨论的那样，事件驱动编程主要是为了解决 I/O 问题。当处理不需要 I/O 操作的内存数据时，Node 也可以使用完全过程式结构。

I/O问题

我们先从高效系统需要用到的 I/O 类型开始入手，这些是所用到的模式的基础。

首先要查看的是串行与并行 I/O 间的明显区别。串行很简单，即先进行一个 I/O 操作，完成以后，再做下一个；并行实现起来比较复杂，但也不难理解，即同时进行两个 I/O 操作。重点是，在串行任务中通常完成顺序是确定的，而在并行任务中任何一个操作先完成返回都是有可能的。

串行和并行任务混合在一起也是可以的。例如，两组并行请求可以串行执行，即先同时执行两个任务，然后再同时执行另外两个任务。

在 Node 里，我们假设所有的 I/O 都有无限长的延迟，也就是说任何 I/O 任务都可能需要花费 0 到无限长的时间。我们不清楚，也无法假设，这些任务会执行多长时间，所以我们不会等待它们执行，而是使用占位器（事件），它会在 I/O 完成时触发回调函数。因为我们假设了无限延迟条件，这让执行并行任务变得简单。你只需要对不同的 I/O 操作进行调用，它们会在准备好数据的时候返回给你，但是次序是无法预测的。按顺序的串行请求可以用嵌入的方法，或者把引用回调放在一起。这样当第一个回调触发时，它会发起第二个 I/O 请求，第二个回调则启动第三个，以此类推。即使所有的请求都是匿名的，而且不会堵塞事件循环，它们还是会以串行顺序执行。这种按次序执行的模式是有必要的，比如下一个 I/O 操作需要依赖第一个 I/O 操作的结果。

到目前为止，我们有两种方法来操作 I/O：按顺序的串行请求，无序的并行请求。有序的并行请求也是一种有用的模式，比如当我们允许 I/O 操作同时发出请求，但又需要按特定的次序来处理结果时。无序的串行 I/O 操作并没有什么用处，所以我们不把它作为一种模式使用。

1. 无序的并行 I/O

我们先从无序的并行 I/O 开始（例 3-3），因为这是在 Node 中最容易实现的。事实上，Node 中所有的 I/O 操作默认都是无序并行的，因为 Node 的所有 I/O 都是异步非阻塞的。我们操作 I/O 时，只要扔出请求然后等待结果就行了。所有请求可能按

我们操作的顺序执行，也可能不是。我们指的无序，并不是指乱序，而是指顺序没有保证。

例 3-3　Node 中的无序并行 I/O

```
fs.readFile('foo.txt', 'utf8', function(err, data) {
  console.log(data);
};
fs.readFile('bar.txt', 'utf8', function(err, data) {
  console.log(data);
};
```

简单地调用 I/O 请求并指定回调函数就会创建无序并行 I/O 操作。在未来的某一时刻，所有的这些回调函数都会被触发，但哪一个先被触发是未知的。而且，如果某一个请求返回了错误而非数据，也不会影响其他请求。

2. 顺序串行 I/O

在这个模式里，我们希望按顺序执行一些 I/O（无限延时）任务。每一个任务都必须在上一个任务完成后才能开始。在 Node 里，这意味着使用嵌套回调，这样可以在每个任务的回调函数里发起下一个任务，如例 3-4 所示。

例 3-4　嵌入回调函数来完成顺序请求

```
server.on('request', function(req, res) {
  // 从 memcached 里获取 session 信息
  memcached.getSession(req, function(session) {
    // 从 db 获取信息
    db.get(session.user, function(userData) {
      // 其他 Web 服务调用
      ws.get(req, function(wsData) {
        // 渲染页面
        page = pageRender(req, session, userData, wsData);
        // 输出响应内容
        res.write(page);
      });
    });
  });
});
```

虽然嵌入回调函数很容易创建顺序串行 I/O，但它的代码看起来很像“金字塔”[1]。这样的代码很难阅读和理解，也难以维护。比如，扫一眼例 3-4 并不能看清楚 `memcached.getSession` 请求完成后发起 `db.get` 请求，等 `db.get` 完成后又发起 `ws.get` 请求，等等。要让代码可读又不会破坏顺序串行模式，有几种方法。

第一，我们可以继续用内联函数声明，但要给它们增加名字（如例 3-5）。这样容易调试，而且还表明了该回调函数的目的。

注 1：这个词是由 Tim Caswell 杜撰的。

例 3-5　回调函数中的命名函数

```
server.on('request', getMemCached(req, res) {
  memcached.getSession(req, getDbInfo(session) {
    db.get(session.user, getWsInfo(userData) {
      ws.get(req, render(wsData) {
        // 渲染页面
        page = pageRender(req, session, userData, wsData);
        // 输出响应内容
        res.write(page);
      });
    });
  });
});
```

另一种方法需要改变代码风格，用提前声明的函数代替匿名函数或命名函数。这会把金字塔拆散，改为按执行顺序展示，并且代码被拆分成更加可控的小块（请参考例 3-6）。

例 3-6　用声明函数把代码分离

```
var render = function(wsData) {
  page = pageRender(req, session, userData, wsData);
};

var getWsInfo = function(userData) {
  ws.get(req, render);
};

var getDbInfo = function(session) {
  db.get(session.user, getWsInfo);
};

var getMemCached = function(req, res) {
  memcached.getSession(req, getDbInfo);
};
```

例子中的代码并不能真的工作。原本的嵌套代码利用闭包封装了一些变量，使得内嵌函数能够访问这些变量。所以，当状态不需要在 3 个以上回调函数间共享时，声明式函数比较适合。如果下一个函数只需要从上一个回调函数中获取信息，声明式函数就能很好地工作（特别是加上文档的话），而且比多层嵌套函数可读性更好。

当然，让数据在函数间传递有多种方法。大多数情况下，我们还是采用 JavaScript 语言本身的特性。JavaScript 有函数作用域，意思是当你在函数内部定义变量时，这个变量在函数内本地可见。但是，简单的使用 { 和 } 并不会限制一个变量的作用域。这就允许我们在内嵌的回调函数中可以访问外部回调函数内定义的变量，即使外部函数已经返回并关闭了也依然能够访问。当嵌套回调函数时，我们隐式地把所有之前回调函数中的变量都绑定到最新定义的回调函数内了，这也让许多嵌套变得复杂。

我们依然可以采用展开的重构方法，但需要创建一个把原始请求都包含的共享作用域，用一个闭包把所有的回调函数都包含进去。这样，所有与初始请求相关的回调函数都被封装起来，并通过闭包内的变量共享状态（例 3-7）。

例 3-7　在回调函数中封装

```
        server.on('request', function(req, res) {

    var render = function(wsData) {
      page = pageRender(req, session, userData, wsData);
    };

    var getWsInfo = function(userData) {
      ws.get(req, render);
    };

    var getDbInfo = function(session) {
      db.get(session.user, getWsInfo);
    };

    var getMemCached = function(req, res) {
      memcached.getSession(req, getDbInfo);
    };

  }
```

采用这一做法，不但代码组织更有逻辑性，而且利用展开的方法避免了多层嵌套的困绕。

其他创新的组织方式也是存在的。有时候你可以把代码重用到许多函数中，此时就有了中间件的用武之地了。中间件有许多实现方法，Node 中最流行的一种做法是 Connect 框架所使用的模块，就像是 Ruby 世界中的 Rack 一样。其实现背后的思想是，在传递过程中，不单把状态传递过去，还包括了跟状态交互的方法。

在 JavaScript 中，对象是以引用的方式传递的。意思是，当你调用我的 `Function(someObject)` 时，对 `someObject` 的任何修改，都会影响你当前函数作用域中所有对 `someObject` 的引用。这存在着潜在风险，但只要你小心处理可能出现的副作用，就会收获巨大的能力。副作用在异步代码中是非常危险的。如果回调函数使用的对象被别人修改了，它难以确定是什么时候被修改的，因为运行的次序是非线性的。如果你对参数传递进来的对象进行修改的话，需要仔细考虑这些对象将来会在什么地方使用。

简单的做法是，用某个东西来表示状态，然后把它在所有需要依赖此状态的函数间传递。这就需要所有依赖此状态的函数通过统一的接口来互相传递。这也是 Connect（以及 Express）中间件的形式都是 `fucntion(req, res, next)` 的原因。

关于 Connect/Express 中间件的详细内容我们将在第 7 章讨论。

现在，我们先看看基本思路（例 3-8）。当在函数间共享对象时，调用堆栈上靠前的函数会影响这些对象的状态，并传递给后续函数。

例 3-8　在函数间传递修改后的内容

```
        var AwesomeClass = function() {
  this.awesomeProp = 'awesome!'
  this.awesomeFunc = function(text) {
    console.log(text + ' is awesome!')
  }
}

var awesomeObject = new AwesomeClass()

function middleware(func) {
  oldFunc = func.awesomeFunc
  func.awesomeFunc = function(text) {
    text = text + ' really'
    oldFunc(text)
  }
}

function anotherMiddleware(func) {
  func.anotherProp = 'super duper'
}

function caller(input) {
  input.awesomeFunc(input.anotherProp)
}

middleware(awesomeObject)
anotherMiddleware(awesomeObject)
caller(awesomeObject)
```

3.3　编写产品代码

写书的一个挑战就是如何用最简单的语言把事情讲清楚，但这又与展示能实际部署使用的技术和功能代码背道而驰。虽然我们应该尽力写出最简洁、最易懂的代码，但当你需要做些事情让代码更加健壮或运行更快时，代码就可能不那么简单了。本节指导你加固即将部署的代码，也便于你学习后面章节的内容。我们将介绍如何编写成熟的代码，让程序能够长久地运行。不必说你也能理解，如果编写的代码很健壮，将能够大大避免以后的维护问题。Node 单线程方面的取舍使其倾向于变得脆弱，本节要讲的这些技术则能够帮助减轻这个风险。

部署生产线上的应用和在笔记本上运行测试代码是不一样的。服务器有许多不同的资源限制，但通常拥有比开发机器更多的资源。比如说，与笔记本或台式机相比，

前端服务器有更多的 CPU、更大的 RAM，硬盘空间却小得多。Node 现在有一些限制，比如规定了最大的 JavaScript 堆栈大小。这会影响你的部署方式，因为在使用 Node 的易编程、单线程模型来部署时，需要考虑如何充分地利用机器的 CPU 和内存。

3.3.1 差错处理

在本章前面的内容里，我们介绍了 Node 如何把 I/O 操作与其他活动分离开，差错处理也是类似的行为。JavaScript 包含了 try/catch 功能，但这个方法只有当错误发生在内联位置时才有用。使用 Node 的非阻塞 I/O 时，你给函数传递了一个回调函数，这意味着回调函数被事件触发调用时，是不在 try/catch 代码块中的。我们需要为异步运行情景提供差错处理的方法。看看例 3-9 的代码。

例 3-9 尝试在回调函数中捕获错误但失败了

```
var http = require('http')

var opts = {
  host: 'sfnsdkfjdsnk.com',
  port: 80,
  path: '/'
}

try {
  http.get(opts, function(res) {
    console.log('Will this get called?')
  })
}
catch (e) {
  console.log('Will we catch an error?')
}
```

当调用 `http.get()` 时，实际会发生什么呢？我们传入了一些参数让 I/O 进行指定的操作，同时还绑定了回调函数。当 I/O 操作完成时，回调函数会被调用，但是，`http.get()` 在设置好回调函数后，就直接完成并继续运行下去了。如果在 GET 过程中发生错误，将不会被 try/catch 捕获。

I/O 错误的隔离在 Node 命令行解析器中更为明显，因为变量返回时如果没有赋值，命令行会打印出这个变量。我们可以看到 `http.get()` 函数的返回变量是新创建的 `http.ClientRequest` 对象，这就表示 try/catch 完成了它的工作，保证了特定的代码返回时没有发生错误。但是，因为 hostname 是不存在的，将会在 I/O 请求时出错，也就是回调函数不会成功调用。try/catch 并不能解决此问题，因为错误发生在这个 JavaScript 代码外面。当 Node 遇到错误想要报告时，我们早已不在那个栈上了。我们已经在处理另一个事件。

在 Node 中，我们利用 `error` 事件来处理此问题。这是一个特殊的事件，当错误发生时它就会触发。这让参与 I/O 的模块触发另外一个事件给负责处理错误的回调函数。`error` 事件让我们能够处理所有使用的模块中可能出现的问题。让我们以正确方式写一下前面的例子，如例 3-10 所示。

例 3-10　通过 `error` 事件捕捉 I/O 错误

```
var http = require('http')

var opts = {
  host: 'dskjvnfskcsjsdkcds.net',
  port: 80,
  path: '/'
}

var req = http.get(opts, function(res) {
  console.log('This will never get called')
})

req.on('error', function(e) {
  console.log('Got that pesky error trapped')
})
```

通过使用 `error` 事件，我们可以处理对应的错误（在本例中是忽略错误）。最重要的是，我们的程序存活下来了。就像 JavaScript 的 try/catch 那样，`error` 事件捕获了所有类型的异常。一种更好的常用异常处理方法是，对已知的错误条件设置好检查条件，并尽可能处理它们。此外，捕获剩余的错误，记录下来，并保持你的服务器继续运行也许是最佳的方法。

3.3.2　使用多处理器

我们说过，Node 是单线程的，这意味着 Node 只能利用一个处理器来工作。但是，多数服务器都有多个“多核”处理器，一个多核处理器就包含了几个处理器。有两个物理 CPU 插槽的服务器可能有 24 个逻辑核，也就是说操作系统看起来有 24 个处理器。要充分发挥 Node 的作用，需要把这些处理器都利用起来。但如果没有多线程，该如何做呢？

Node 提供了一个 `cluster` 模块，可以把任务分配给子进程，就是说 Node 把当前程序复制了一份给另一个进程（在 Windows 上，它其实是另外一个线程）。每个子进程有些特殊的能力，比如能够与其他子进程共享 socket 连接。这样我们就可以写一个 Node 程序，让它创建许多其他 Node 程序，并把任务分配给它们。

需要重点理解的是，当你用 `cluster` 把工作共享到一组复制的 Node 程序时，主进程不会参与到每个具体的事务中。主进程管理所有的子进程，但当子进程与 I/

O 操作交互时，它们是直接进行操作的，不需要通过主进程。这意味着，如果你用 `cluster` 来创建一个 Web 服务器，请求将不会通过你的主进程，而是直接连接到子进程。而且，调度这些请求并不会导致系统出现瓶颈。

通过 cluster API，你可以把工作分配给 Node 进程，并分布在服务器所有可用的处理器上，这能够充分利用资源。让我们看一个简单的 `cluster` 代码例子（例 3-11）。

例 3-11　使用集群来分发任务

```
var cluster = require('cluster');
var http = require('http');
var numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  // 创建工作进程
  for (var i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('death', function(worker) {
    console.log('worker ' + worker.pid + ' died');
  });
} else {
  // 工作进程创建 http 服务器
  http.Server(function(req, res) {
    res.writeHead(200);
    res.end("hello world\n");
  }).listen(8000);
}
```

在例子中，我们使用了 Node 的一些核心模块来把工作平均分配到所有可用的 CPU 上，这些模块为：`cluster` 模块、`http` 模块和 `os` 模块。从 `os` 模块中，我们可以轻松得到系统 CPU 的数量。

`cluster` 工作的原理是每一个 Node 进程要么是主进程，要么成为工作进程。当一个主进程调用 `cluster.fork()` 方法时，它会创建与主进程一模一样的子进程，除了两个让每个进程可以检查自己是父 / 子进程的属性以外。在主进程中（Node 运行时直接调用的那个脚本），`cluster.isMaster` 会返回 `true`，而 `cluster.isWorker` 会返回 `false`。而在子进程，`cluster.isMaster` 返回 `false`，且 `cluster.isWorker` 返回 `true`。

例子中的主脚本为每个 CPU 创建了一个工作进程。每个子进程创建了一个 HTTP 服务器，这是 `cluster` 另一个独特的地方。在使用 `cluster` 的地方使用 `listen()` 监听一个 socket 的时候，多个进程可以同时监听同一个 socket。如果通过调用 `node myscript.js` 的方法启动多个 Node 进程，会导致出错，因为第二个进程在启动时

会抛出 EADDRINUSE 的异常。cluster 提供了跨平台时让多个进程共享 socket 的方法。即使多个子进程在共享一个端口上的连接，其中一个堵塞了，也不会影响其他工作进程的新连接。

除了共享 socket 外，我们还能用 cluster 做更多事情，因为它是基于 child_process 模块的。这个模块会提供一系列属性，其中最有用的一些可以检查子进程健康状态。在上面的例子中，当子进程死亡时，主进程会用 console.log() 输出死亡提醒。例 3-12 中提供了一个更实用的例子，它会调用 cluster.fork() 来创建一个新的子进程。

例 3-12　出现死亡进程后重新开启新的进程

```
      if (cluster.isMaster) {
  // 创建工作进程
  for (var i=0; i<numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('death', function(worker) {
    console.log('worker ' + worker.pid + ' died');
    cluster.fork();
  });
}
```

这个简单的改造让主进程会不停地把死掉的进程重启，从而保证所有的 CPU 都有我们的服务器在运行。然而，这只是对运行状态的基本检查，我们还能用更多花哨的技巧。因为工作进程可以传消息给主进程，所以可以让每个工作进程报告自己的状态，如内存使用量。这让主进程可以觉察哪些工作进程变得不稳定，确认哪些工作进程没有冻结，或者被长时间运行的事件堵塞（例 3-13）。

例 3-13　通过消息传递来监控工作进程状态

```
var cluster = require(''cluster');
var http = require('http');
var numCPUs = require('os').cpus().length;

var rssWarn = (12 * 1024 * 1024)
  , heapWarn = (10 * 1024 * 1024)

if(cluster.isMaster) {
  for(var i=0; i<numCPUs; i++) {
    var worker = cluster.fork();
    worker.on('message', function(m) {
      if (m.memory) {
        if(m.memory.rss > rssWarn) {
          console.log('Worker ' + m.process + ' using too much memory.')
        }
      }
```

```
      })
    }
  } else {
    // 服务器
    http.Server(function(req,res) {
      res.writeHead(200);
      res.end('hello world\n')
    }).listen(8000)
    // 每秒报告一次状态
    setInterval(function report(){
      process.send({memory: process.memoryUsage(), process: process.pid});
    }, 1000)
  }
```

在这个例子里，工作进程报告自己的内存使用量，当子进程使用了过多内存时，主进程会发送一条警告到日志中去。这是运维团队常用的检测系统健康状态的功能。这让 Node 主进程有控制的能力，也带来了好处。这个消息传递的接口也允许主进程把消息发回给工作进程，这意味着你可以把主进程当成工作进程的一个轻量级控制接口。

我们还能用消息传递做更多的事情，而这些事情无法在 Node 之外实现。因为 Node 依赖事件循环来工作，所以有个风险是其中一个事件回调函数运行了很长的时间，这会导致该进程的其他用户需要等待很长时间才能得到服务。主进程与每个工作进程有一个连接，所以我们可以告诉它定时发送"all OK"消息，这样我们就能够验证事件循环在以合适的速度周转着，并没有被某个回调函数堵塞。可悲的是，即使识别了一个长时间运行的回调函数，我们也无法主动关闭它。因为我们发送给该进程的任何通知都会加到事件队列里，所以它需要等待已经在长时间运行的回调函数结束后才会被处理。因此，虽然我们能够让主进程识别僵尸进程，但唯一的补救方法就是杀掉工作进程，而这会丢失它正在执行的工作。

做些准备工作就能让你有能力杀掉某个威胁到系统资源的工作进程，如例 3-14 所示。

例 3-14　杀死僵尸进程

```
var cluster = require('cluster');
var http = require('http');
var numCPUs = require('os').cpus().length;

var rssWarn = (50 * 1024 * 1024)
  , heapWarn = (50 * 1024 * 1024)

var workers = {}

if(cluster.isMaster) {
  for(var i=0; i<numCPUs; i++) {
```

```
    createWorker()
  }

  setInterval(function() {
    var time = new Date().getTime()
    for(pid in workers) {
      if(workers.hasOwnProperty(pid) &&
         workers[pid].lastCb + 5000 < time) {

        console.log('Long running worker ' + pid + ' killed')
        workers[pid].worker.kill()
        delete workers[pid]
        createWorker()
      }
    }
  }, 1000)
} else {
  //服务器
  http.Server(function(req,res) {
    //打乱 200 个请求中的 1 个
    if (Math.floor(Math.random() * 200) === 4) {
      console.log('Stopped ' + process.pid + ' from ever finishing')
      while(true) { continue }
    }
    res.writeHead(200);
    res.end('hello world from '  + process.pid + '\n')
  }).listen(8000)
  //每秒钟报告一次状态
  setInterval(function report(){
    process.send({cmd: "reportMem", memory: process.memoryUsage(),
process: process.pid})
  }, 1000)
}

function createWorker() {
  var worker = cluster.fork()
  console.log('Created worker: ' + worker.pid)
  //允许开机时间
  workers[worker.pid] = {worker:worker, lastCb: new Date().getTime()-1000}
  worker.on('message', function(m) {
    if(m.cmd === "reportMem") {
      workers[m.process].lastCb = new Date().getTime()
      if(m.memory.rss > rssWarn) {
        console.log('Worker ' + m.process + ' using too much memory.')
      }
    }
  })
}
```

在这个脚本中，我们给主进程也添加了类似工作进程的定时器。现在，每当一个工作进程向主进程发送报告时，主进程都会记录报告的时间。大约每隔一秒，主进程就会检查所有的工作进程，看看是否有某个进程已经超过 5 秒未更新状态（因为超时是以微秒为单位，所以我们用的是 `>5000`）。如果发现这样的进程，主进程将把阻

塞的工作进程杀掉并重启。为了让这个流程更加高效，我们把创建工作进程的代码放到一个小程序里，这样就能在同一个地方为不同情景提供启动工作，无论是创建新的工作进程还是重启死亡进程。

我们也对 HTTP 服务器做了一个小改动，让每个请求有 1/200 的概率会出错。你可以运行一下脚本，看看出现错误的可能。如果你同时从多个地方发起并行请求，就能看到整个代码是如何运行的。这些彻底分隔的 Node 程序通过消息传递来进行交互。因为主进程是简单的小程序，不会卡住，所以它在任何情况下都能够一直检查其他进程。

第二部分

API和常用模块

第 4 章

核心API

Node 提供了许多 API，其中一些比较重要。这些核心 API 是所有 Node 应用的支柱，你会不停地用到它们。

4.1　Events

我们第一个要研究的 API 是 Events API。这是因为，尽管抽象，但它是其他所有 API 工作的基础模块。通过仔细查看这个 API，你就能很好地使用其他所有的 API。

如果你曾经在浏览器里开发过 JavaScript 程序，就一定已经使用过 Events 了。但是，浏览器的事件模型是从 DOM 里来的，不是 JavaScript 本身自带的。DOM 的许多理念在其使用情景之外并没有太多用处。我们先看看 DOM 模型的 Events，然后再与 Node 的实现方式进行比较。

DOM 是基于用户交互的用户驱动型事件模型，有着一组与树状结构（HTML，XML，等等）对应的接口元素。意思是，当用户与接口的某个特定部分交互时，对应有一个事件和一个相关的对象，比如某个 HTML/XML 元素被点击或者进行了其他操作。该操作对象有父节点，并且可能有子节点。因为操作对象是在一棵树中，所以这个模型包含了冒泡和捕获的概念，就是允许沿着树的结构向上或向下的元素也接收被触发的事件。

例如，在一个 HTML 列表中，在 `<li>` 上的一个点击事件，能够被其父节点 `<ul>` 上绑定的监听器所捕获。反过来，`<ul>` 上的点击会向下冒泡传给 `<li>` 上的监听

器。因为 JavaScript 对象没有这一类树状结构，所以 Node 中的模型更加简单。

4.1.1 EventEmitter

因为在浏览器中 Event 模型是绑定在 DOM 上的，所以 Node 创建了 EventEmitter 类来提供基础的事件功能。所有 Node 的事件功能围绕着 EventEmitter，因为它的设计包含了其他类扩展所需要的接口类。EventEmitter 对象通常不会直接调用。

EventEmitter 类提供了一系列方法，其中最主要的两个是 on 和 emit，这些方法供其他类使用。on 方法为一个事件创建了监听器，如例 4-1 所示。

例 4-1　使用 on 方法监听事件

```
server.on('event', function(a, b, c) {
  //具体操作
});
```

on 方法接受两个参数：需要监听的事件的名称，当事件触发时需要调用的函数。因为 EventEmitter 是接口，从 EventEmitter 继承的类需要使用 new 关键字来构造。让我们看看例 4-2 中如何构造一个监听器的新类。

例 4-2　创建一个新类支持 EventEmitter 事件

```
var utils = require('utils'),
    EventEmitter = require('events').EventEmitter;

var Server = function() {
  console.log('init');
};

utils.inherits(Server, EventEmitter);

var s = new Server();

s.on('abc', function() {
  console.log('abc');
});
```

例子中，我们先包含了 utils 模块，以便调用它的 inherits 方法。inherits 能够把 EventEmitter 类的方法添加到我们创建的 Server 类中。这意味着所有 Server 的新实例都能够使用 EventEmitter 的方法。

然后我们包含了 events 模块。但我们只想调用其模块中的 EventEmitter 类。注意，EventEmitter 是以大写字母开始命名的，用来表示它本身是一个类。我们不调用 createEventEmitter 方法，因为不打算直接使用 EventEmitter 实例，只是想把它的方法绑定到我们要用的 Server 类上。

当包含了所有需要的模块后，下一步就是创建基础的 `Server` 类。它只提供了一个简单的函数，就是在初始化的时候记录一条消息。在真实的实现中，我们会为 `Server` 类补充它需要使用的函数原型。为了简单起见，我们先把这部分省略了。重要的一步是使用 `sys.inherits` 把 `EventEmitter` 作为超类添加给 `Server` 类。

当需要使用 `Server` 类时，我们用 `new Server()` 来实例化它。`Server` 的实例能够访问其超类（`EventEmitter`）的方法，也就是说我们可以调用 on 方法来为这个实例添加监听器。

到目前为止，我们添加的事件监听器还不会被调用，因为并没有 `abc` 事件被触发。我们可以通过添加例 4-3 中的代码来触发这个事件。

例 4-3　触发一个事件

```
s.emit('abc');
```

触发事件监听器很简单，只要调用从 `EventEmitter` 继承的 `Server` 实例的 emit 方法就行了。需要注意的是，这些事件是针对某个实例的，不存在全局的事件。当你调用 on 方法的时候，需要绑定在特定的基于 `EventEmitter` 的对象上。`Server` 类不同的实例之间也不会共享事件。例 4-3 代码中的 `s` 对象不会与另外一个 `Server` 实例 `z`（比如 `var z = new Server()`）共享同一个事件。

4.1.2　Callback语法

使用事件很重要的一个部分是处理回调函数。第 3 章已经深入地探讨了其最佳实践，但我们现在要看看在 Node 里回调函数的工作机制。它们采用了几种标准模式，我们先来看看有哪些可能性。

当调用 `emit` 时，除了事件的名称，你可以传入任意数目的参数。例 4-4 中包含了 3 个这样的参数。这些参数都将传给该监听该事件的函数。比如，从 `http` 服务器接收到 `request` 请求时，你会收到两个参数：`req` 和 `res`。当 `request` 事件被触发时，这些参数会作为第二个和第三个参数传给 `emit` 函数。

例 4-4　触发事件的时候传递参数

```
s.emit('abc', a, b, c);
```

了解 Node 如何调用事件监听器很重要，因为这会影响你的编程风格。当 `emit()` 调用包含变量时，例 4-5 中的代码会被用来调用对应的事件监听器。

例 4-5　触发器里如何调用事件

```
if (arguments.length <= 3) {
```

```
    // 速度快
    handler.call(this, arguments[1], arguments[2]);
  } else {
    // 速度慢
    var args = Array.prototype.slice.call(arguments, 1);
    handler.apply(this, args);
  }
```

这两种代码都是 JavaScript 调用函数的方法。如果传给 `emit()` 的参数只有 3 个或更少，该方法就会使用捷径，直接调用 `call` 方法。否则，它就会使用较慢的 `apply` 方法，以数组的方式传递所有的参数。这里需要注意的是，Node 调用这两种方法时都直接使用了 `this` 参数。这意味着事件监听器被调用的时候是在 `EventEmitter` 的上下文中，而不是它们原始的位置。通过 Node 命令行解析器，你可以清楚地看见当 `EventEmitter` 调用对象时会发生什么事情（例 4-6）。

例 4-6　`EventEmitter` 改变了上下文

```
> var EventEmitter = require('events').EventEmitter,
...     util = require('util');
>
> var Server = function() {};
> util.inherits(Server, EventEmitter);
> Server.prototype.outputThis= function(output) {
...   console.log(this);
...   console.log(output);
... };
[Function]
>
> Server.prototype.emitOutput = function(input) {
...   this.emit('output', input);
... };
[Function]
>
> Server.prototype.callEmitOutput = function() {
...   this.emitOutput('innerEmitOutput');
... };
[Function]
>
> var s = new Server();
> s.on('output', s.outputThis);
{ _events: { output: [Function] } }
> s.emitOutput('outerEmitOutput');
{ _events: { output: [Function] } }
outerEmitOutput
> s.callEmitOutput();
{ _events: { output: [Function] } }
innerEmitOutput
> s.emit('output', 'Direct');
{ _events: { output: [Function] } }
Direct
true
>
```

输出例子中首先设置了一个 `Server` 类，它包含了触发 `output` 事件的函数。`outputThis` 方法作为事件监听器绑定在 `output` 事件上。在不同的上下文中触发 `output` 事件时，我们保持在 `EventEmitter` 对象所在的作用域中。所以 `s.outputThis` 能够访问的 `this` 变量的值是属于该 `EventEmitter` 的。因此，如果我们想在事件回调函数中使用 `this` 变量的话，就必须把它作为一个参数传入并赋值到另外一个变量上。

4.2 HTTP

Node.js 的核心功能之一就是作为 Web 服务器，这是这个系统的一个重要部分，所以当 Ryan Dahl 发起此项目时，他为 V8 重写了 HTTP 模块，使其能够非阻塞运行。虽然最开始的 HTTP 实现已经蜕变了许多，API 和内部实现不断升级，但核心操作还是保持不变的。Node 实现的 HTTP 模块是非阻塞的，且速度很快，其中许多代码已经从 C 迁移到了 JavaScript。

HTTP 使用的模式在 Node 里很常见。父工厂类提供了很容易创建新服务器的方法 [1]。`http.createServer()` 方法为我们提供了构建新的 HTTP`Server` 类的实例。我们可以为新的类定义 Node 接受到 HTTP 请求时的操作。HTTP 模块及其他 Node 模块还有一些共性的地方，比如 `Server` 类能够触发的事件，还有传给回调函数的数据结构。了解这 3 种类型有助于你更好地使用 HTTP 模块。

4.2.1 HTTP服务器

HTTP 服务器也许是 Node 最常用的使用情景了。在第 1 章中，我们设置了一个 HTTP 服务器，并用它来处理非常简单的请求。其实，HTTP 还有很多的功能。HTTP 模块的服务器部分提供了构建复杂、全面的 Web 服务器的原始工具。在本章里，我们会继续探索处理请求及发送响应的机理。即使你使用了更高级的服务器框架（如 Express），它们背后采用的许多概念也都是从这里介绍的内容延伸而来的。

正如前面的例子所示，使用 HTTP 服务器的第一步就是调用 `http.createServer()` 方法来创建一个新的服务器。这会返回一个新的 `Server` 类的实例，里面只包含少数的方法，因为更多的功能是通过使用事件来提供的。`http` 服务器类有 6 个事件和 3 个方法。还要注意的是，大部分方法只是用来初始化服务器，而事件则是用来处理它的运行时操作。

让我们从创建一个最小的 HTTP 服务器代码开始（例 4-7）。

注 1：当我们提到父类(pseudoclass)时，指的是 Douglas Crockford 所著的 JavaScript: The Good Parts（O'Reilly）一书中的定义。从现在起，我们将使用“类”的说法来代替“父类”。

例 4-7　非常简短的 HTTP 服务器

```
require('http').createServer(function(req,res){res.writeHead(200, {});
res.end('hello world');}).listen(8125);
```

这个例子并不是好的代码风格，但是它演示了几个关键点，稍后我们再改进代码风格。首先，我们用 require 包含了 http 模块。注意我们使用了链式方法来使用这个模块，而不需要先赋值给一个变量。Node 里的许多东西会返回一个函数[2]，这样我们就能直接调用这些函数了。包含 http 模块后，我们调用 createServer。这里并不是必须输入参数，但我们传入了一个函数，并让它绑定在 request 事件上。最后，我们让 createServer 创建的服务器监听（listen）8125 端口。

希望你在真实情景下从不需要写类似的代码，但这里的代码显示了语法的灵活和语言的简洁。让我们把代码写得更清晰些。例 4-8 中重写的代码变得更好理解和维护了。

例 4-8　简短但可读性更好的 HTTP 服务器

```
var http = require('http');
var server = http.createServer();
var handleReq = function(req,res){
  res.writeHead(200, {});
  res.end('hello world');
};
server.on('request', handleReq);
server.listen(8125);
```

这个例子依旧是实现了最小的 Web 服务器，但这次，我们开始把东西赋值给命名变量。比起使用链式调用，这让代码变得更容易读一些，而且还可以重用。例如，在一个文件里可能会多次使用 http，这样的情形并不少见。如果你想同时使用 HTTP 服务器和 HTTP 客户端，重用该模块对象就会很有用。即使 JavaScript 并不逼你去考虑内存，也不意味着你可以把不需要的对象草率地到处乱扔。然后我们还是用了命名函数来处理 request 事件，用来替换之前的匿名回调函数。这与内存使用没什么关系，只是增加了可读性。我们并不是说你不该使用匿名函数，但如果可以把代码放在容易查找的地方，会更有助于维护它。

> 阅读本书的第一部分可以更好地掌握编程风格，特别是第 1 章和第 2 章中关于代码风格的处理。

因为我们没有在调用创建 http 服务器对象的工厂方法时传入 request 事件监听器，所以需要显式地添加该事件监听器，这可以利用 EventEmitter 调用 on 方法来完成。最后，和之前的例子一样，我们调用 listen 方法来监听想要的端口。http 类

注 2：这是因为 JavaScript 支持一等函数（first-class functions）。

还提供了其他函数，这个例子只是演示了最重要的几个。

http 服务器支持几种事件，都是和客户端的 TCP 或者 HTTP 连接相关联的。connection 和 close 事件表示了与客户端的 TCP 连接的建立与关闭。要注意，如果客户端使用的是 HTTP 1.1 协议，这是支持 keepalive 的。这意味着这些 TCP 连接可能会跨越多个 HTTP 请求。

request、checkContinue、upgrade 和 clientError 事件是关联在 HTTP 请求上的。我们已经使用过 request 事件，表示的是新的 HTTP 请求。

checkContinue 事件是一个特殊事件，当客户端以数据流的方式将数据发送给服务器时，你可以对 HTTP 请求进行更直接的控制。客户端发送数据给服务器时，需要检查当前状态下能否继续，这就会触发此事件。如果这个事件绑定了事件处理器，则 request 请求就不会被触发。

upgrade 事件在一个客户端请求协议升级时会触发。除非绑定了此事件的事件处理器，否则 http 服务器将拒绝 HTTP 升级请求。

最后，clientError 事件会把客户端发送的 error 事件传递出来。

HTTP 服务器可以抛出若干事件，最常见的是 request，但你也会在 TCP 连接的整个生命周期中获得其他事件。

当为请求创建一个新的 TCP 流时，就会触发 connection 事件。这个事件会把 TCP 流作为参数传给该请求。该数据流也可以在 request 使用的时候，通过 request.connection 变量获得。但每个流只会触发 connection 事件一次，所以可能会出现从一个客户端来的多个请求只对应一次 connection 事件的情况。

4.2.2 HTTP客户端

如果你想向远程服务器发起 HTTP 连接，Node 也是很好的选择。Node 在许多情景下都很适合使用，如使用 Web service，连接到文档数据库，或是抓取网页。你可以使用同样的 http 模块来发起 HTTP 请求，但应该使用 http.ClientRequest 类。该类有两个工厂方法：一个通用的方法和一个便捷的方法。让我们看看例 4-9 的通用方法的例子。

例 4-9　创建 HTTP 请求

```
var http = require('http');

var opts = {
  host: 'www.google.com'
```

```
  port: 80,
  path: '/',
  method: 'GET'
};

var req = http.request(opts, function(res) {
  console.log(res);
  res.on('data', function(data) {
    console.log(data);
  });
});

req.end();
```

你首先要查看的是配置（options）对象，它定义了请求的许多功能。我们必须提供 host 名字（虽然 IP 地址也是可以的）、端口（port）和路径（path）。方法（method）是可选项，如果没有指定，默认会设置为 GET。在本质上，这个例子指定了往 http://www.google.com/ 的 80 端口发起 HTTP GET 请求。

接下来，我们要用配置（options）对象来创建一个 http.ClientRequest 实例，就是调用 http.request() 这个工厂方法，并传入 options 对象和回调函数（可选）。传入的回调函数会监听 response 事件，并在接收到 response 事件时，处理 request 的数据。在之前的例子里，我们只是简单地把 response 对象打印到终端上。但要注意一点，HTTP 请求的正文内容实际上是通过 response 对象的数据流获得的。而且你可以订阅 response 对象的 data 事件，以便于数据可用时就能处理（更多信息请参见本书 4.3.1 节的内容）。

最后需要注意的一点是，需要结束（end()）该请求。因为这是一个 GET 请求，所以我们并不会往服务器发送任何数据。但对于其他的 HTTP 方法，比如 PUT 或 POST，你可能需要发送数据。request 会等待 end() 方法调用后，才初始化 HTTP 请求，因为在那之前，它不确定我们是否还会发送数据。

1. 提交 HTTP GET 请求

GET 是很常见的 HTTP 使用方式，因此提供了一个专门的工厂方法来更方便地使用它，如例 4-10 所示。

例 4-10　简单的 HTTP GET 请求

```
var http = require('http');

var opts = {
    host: 'www.google.com'
    port: 80,
    path: '/',
};
```

```
var req = http.get(opts, function(res) {
    console.log(res);
    res.on('data', function(data) {
      console.log(data);
    });
});
```

这个例子的 `http.get()` 和之前的例子做了一样的事情，但更加明确。我们把 `method` 属性从配置对象中去掉了，还把 `request.end()` 也移除了，因为这些都已经隐含说明了。如果运行了这两个例子，你得到的结果将是 `Buffer` 对象的裸数据。本章后续会介绍到，`Buffer` 是 Node 特殊定义的类，用来支持任意二进制数据的存储。虽然你也可以直接使用这些内容，但通常要指定编码方式，如 UTF-8（一种 Unicode 字符的编码格式），这可以通过 `response.setEncoding()` 方法来指定（例 4-11）。

例 4-11　对比裸 `Buffer` 输出与指定编码格式的输出

```
> var http = require('http');
> var req = http.get({host:'www.google.com', port:80, path:'/'},
function(res) {
... console.log(res);
... res.on('data', function(c) { console.log(c); });
... });
> <Buffer 3c 21 64 6f 63 74 79 70

...

65 2e 73 74>
<Buffer 61 72 74 54 69

...

69 70 74 3e>

>
> var req = http.get({host:'www.google.com', port:80, path:'/'},
function(res) {
... res.setEncoding('utf8');
... res.on('data', function(c) { console.log(c); });
... });
> <!doctype html><html><head><meta http-equiv="content-type

...

load.t.prt=(f=(new Date).getTime());
})();
</script>

>
```

在第一个例子中，我们没有调用 `ClientResponse.setEncoding()`，而且得到的是 `Buffer` 中的块数据。虽然输出是简略的打印内容，但也能看出并非只有一个

Buffer，而是分开了几个 Buffer 才把数据返回完整。在第二个例子中，因为我们设置了 res.setEncoding('utf8')，数据以 UTF-8 的格式返回了。 从服务器返回的数据还是一样，分成了几块，但这次发给程序的是以正确的编码显示的字符串，而不再是 Buffer 的裸数据。虽然打印的内容可能表现得还不够明显，但每个原始 Buffer 对应打印出来的都是一个字符串。

2. 发送 HTTP POST 和 PUT 数据

不是所有的 HTTP 请求都是用 GET 方法的，你还需要调用 POST、PUT 和其他 HTTP 方法，它们会改变对方的数据。这和发送 GET 请求的功能一样，只不过你还需要往上发送一些数据（例 4-12）。

例 4-12　往上传服务写入数据

```
var options = {
  host: 'www.example.com',
  port: 80,
  path: '/submit',
  method: 'POST'
};

var req = http.request(options, function(res) {
  res.setEncoding('utf8');
  res.on('data', function (chunk) {
    console.log('BODY: ' + chunk);
  });
});

req.write("my data");
req.write("more of my data");

req.end();
```

这个例子和例 4-10 很相似，但增加了 http.ClientRequest.write() 方法。可以用这个方法发送上行数据流。之前解释过，它要求你显式地调用 http.ClientRequest.end() 方法来表示数据发送完毕。每当调用 ClientRequest.write() 时，数据会马上上传（不会被缓存），但服务器在 ClientRequest.end() 调用之前是不会响应你的数据请求的。

你可以把一个流（Stream）的 data 事件和 ClientRequest.write() 绑定在一起，这样就能把数据以流的形式发送给服务器了。比如当需要把硬盘上的一个文件通过 HTTP 发送给远程服务器时，这会是个好主意。

3. ClientResponse 对象

ClientResponse 对象保存了关于请求的许多信息，而且都很直观。它的一些显著

的属性也很有用，包括 statusCode（包含了 HTTP 状态）和 header 属性（响应头对象）。ClientResponse 上还挂了多个数据流和属性，你也许会直接使用它。

4.2.3 URL

URL 模块提供了解析和处理 URL 字符串的便利工具，当你需要和 URL 打交道时会非常有用。该模块提供了 3 个方法：parse、format 和 resolve。我们先从例 4-13 开始，在 Node 命令行里演示如何使用 parse。

例 4-13 用 URL 模块解析 URL

```
> var URL = require('url');
> var myUrl = "http://www.nodejs.org/some/url/?with=query&param=that&are=
awesome#alsoahash";
> myUrl
'http://www.nodejs.org/some/url/?with=query&param=that&are=awesome
#alsoahash'
> parsedUrl = URL.parse(myUrl);
{ href: 'http://www.nodejs.org/some/url/?with=query&param=that&are=
awesome#alsoahash'
, protocol: 'http:'
, slashes: true
, host: 'www.nodejs.org'
, hostname: 'www.nodejs.org'
, hash: '#alsoahash'
, search: '?with=query&param=that&are=awesome'
, query: 'with=query&param=that&are=awesome'
, pathname: '/some/url/'
}
> parsedUrl = URL.parse(myUrl, true);
{ href: 'http://www.nodejs.org/some/url/?with=query&param=that&are=
awesome#alsoahash'
, protocol: 'http:'
, slashes: true
, host: 'www.nodejs.org'
, hostname: 'www.nodejs.org'
, hash: '#alsoahash'
, search: '?with=query&param=that&are=awesome'
, query:
   { with: 'query'
   , param: 'that'
   , are: 'awesome'
   }
, pathname: '/some/url/'
}
>
```

第一件事情当然是要先包含 URL 模块。注意所有的模块名称都是小写的。我们创建的 url 字符串包含了需要被解析的所有部分。解析真的很简单：只要对该字符串调用 URL 模块的 parse 方法。它返回的数据结构代表了解析出来的 URL 的各个部分，

它产生的组成部分如下：

- href
- protocol
- host
- auth
- hostname
- port
- pathname
- search
- query
- hash

href 是原始输入用来解析的完整 URL。protocol 是用在 URL 里的协议（如 http://、https://、ftp:// 等）。host 是 URL 里完整的 hostname。这可以是本地服务器的 hostname，比如打印机服务器，也可以是如 www.google.com 一样完整的域名。它还可能包含了端口（如 8080），或用户名和密码（如 un:pw@ftpserver.com）。hostname 的不同部分会进一步细分到：auth（包含用户证书）、port（单纯是端口）、hostname（包含 URL 的主机名）。重点是，hostname 依然是完整的主机名，包含了顶级域名（如 .com 和 .net 等）和特定的服务器。如果 URL 是 http://sport.yahoo.com/nhl，hostname 不会单独给你顶级域名（yahoo.com）或只给你主机（sport），而是会给你完整的主机名（sport.yahoo.com）。URL 模块并没有能力把 hostname 细分成单独的部分，如域名或顶级域名。

URL 的下一组成员是关于 host 部分后面的所有东西。pathname 是跟在 host 之后的整个文件路径，例如 http://sports.yahoo.com/nhl 的 pathname 就是 /nhl。下一个是 search 部分，保存了 URL 中 HTTP GET 的参数。比如 URL 是 http://mydomain.com/?foo=bar&baz=qux，search 部分对应的是 ?foo=bar&baz=qux。注意它包含了 ?。query 参数和 search 部分类似，它包含二者中的一项，具体要看 parse 被调用的方法。

parse 可以有两个参数：url 字符串，及一个可选的布尔值，用来确定 queryString 是否该用 querystring 模块来解析（下一小节再详细介绍）。如果第二个参数是 false，query 将包含一个与 search 类似的字符串，但去掉了开头的 ?。如果你没有传入第二个参数，那么默认为 false。

URL 的最后一个部分是片段部分（称为 hash）。这是 URL 中在 # 之后的部分。通常，这是用来指向 HTML 页面内的命名锚记（anchor）。比如 http://abook.

com/#chapter2 可能是指向包含整书内容的网页的第 2 章。在这个例子中，hash 部分就包含了 #chapter2。同样，该字符串包含了“#”。有些网站，如 http://tiwtter.com，使用更复杂的片段来做 AJAX 应用，但基本原则是一样的。所以假如用户 mentions 的 Twitter 账号 URL 是 http://twitter.com/#!/mentions，那么它的 pathname 是 /，但 hash 是 #!/mentions。

4.2.4 querystring

querystring 模块是用来处理 query 字符串的简单辅助模块。上一小节已经讨论过，query 字符串是在 URL 尾部编码过的参数。但是如果只是把它当做 JavaScript 字符串来使用时，处理这些参数未免很烦琐。querystring 模块提供了从 query 字符串中轻松提取对象的方法。它的主要功能有 parse 和 decode，还包括一些内部辅助函数，如 escape、unescape、unescapeBuffer、encode 和 stringify。如果你有一个 query 字符串，你可以使用 parse 来把它变成一个对象（例 4-14）。

例 4-14　在 Node 终端里使用 querystring 模块解析查询字符串

```
> var qs = require('querystring');
> qs.parse('a=1&b=2&c=d');
{ a: '1', b: '2', c: 'd' }
>
```

例子中，该类的 parse 方法把 query 字符串转换成为一个对象，其中属性是对应 query 字符串中的关键字和变量值。你还需要注意几件事情。第一，数字是返回成字符串的，并非数字类型。JavaScript 是弱类型语言，用一个数值运算就能够轻松把一个字符串强制转换成数字。但是需要时刻考虑那些无法强制转换的情况。

其次要注意的是，你传入的 query 字符串不能包含 URL 中标记的 ?。一个典型的 URL 例子是 http://www. bobsdiscount.com/?item=304&location=san+francisco。query 字符串以 ? 开始，表示文件路径已经结束。但如果你把 ? 也包含在传进去解析的字符串中，第一个关键字就以 ? 开头了，你肯定不想得到这样的结果。

这个库在许多使用情景下都非常有用，因为除了 URL 外，很多地方会使用到 query 字符串。当你从一个 HTTP POST 发送的内容是 x-formencoded 格式的时候，它也是以 query 字符串的形式呈现的。所有的浏览器厂商都为这一做法制定了标准。默认情况下，HTML 里的 form 都会用这个方式发送数据到服务器上去。

querystring 模块也被 URL 模块用作辅助模块。特别是在解析 URL 的时候，你可以指定 URL 模块把 query 字符串转换成对象返回给你，而不是给你一个简单的字符串。这在上一小节已经详细介绍过了，但 parse 方法是使用了 querystring 模块

来解析的。

querystring 另外一个重要的部分是 encode（例 4-15）。该函数把输入的 key-value 格式的对象转换成 query 字符串的格式。如果你需要使用 HTTP 请求（特别是 POST 数据），这会非常方便。你可以在操作时使用 JavaScript 对象，然后在需要进行数据传输时再轻松地把它编码成需要的格式。所有的 JavaScript 对象都可以使用，但最好是使用的对象只包含需要的数据，因为 encode 方法会把对象所有的属性都添加进来。但是，如果属性的值不是 string、Boolean 或 number 中的一种，它就不能被序列化，返回的内容中关键字（key）对应的值会是空的。

例 4-15　把对象编码成查询字符串

```
> var myObj = {'a':1, 'b':5, 'c':'cats', 'func': function(){console.
log('dogs')}}
> qs.encode(myObj);
'a=1&b=5&c=cats&func='
>
```

4.3　I/O

I/O 是 Node 有别于其他框架的核心模块之一。本节将探索 Node 中提供非阻塞 I/O 的 API。

4.3.1　数据流（stream）

Node 中的许多组件提供了连续输出或可连续处理输入的功能。 为了让这些组件行为一致，stream API 提供了一个抽象的接口。该 API 提供了常用的方法，以及数据流具体实现时需要使用的属性。数据流分为可读、可写和可读写。所有的流都是 EventEmitter 的实例，也就是说可以主动触发事件。

可读的数据流

可读的数据流 API 是一组方法和事件，提供了数据源在发送时访问数据块的功能。基本上，可读数据流是与触发 data 事件相关的。这些事件流就代表了数据的流形式。为了更加可控，数据流还提供了一些功能让你可以配置返回数据的大小和速度。

最基本的流如例 4-16 所示，它从一个文件里把数据分块读取。每当一个新的数据块准备好的时候，它会把数据以变量 data 的形式传给回调函数。在这个例子里，我们只是简单地把数据记录到终端。但在实际使用场景中，你可以把数据以流的形式发送到其他地方，或者把它积攒起来，然后再一并处理。其实，data 事件只提供了访问数据的方法，你需要想出如何处理每次返回的数据块。

例 4-16　创建可读文件流

```
var fs = require('fs');
var filehandle = fs.readFile('data.txt', function(err, data) {
  console.log(data)
});
```

让我们进一步看看一种处理数据流的常用模式。有时候我们需要等待完整的数据都可用后再进行操作，在这种情况下就会用到*数据池模式*（spooling pattern）。我们知道重点是不要让 Node 的事件循环阻塞，所以即使不想在接收到所有数据之前进行下一步处理，也不希望堵塞事件循环。在这种情况下（例 4-17），我们使用数据流来读取数据，但只有在接收到足够的内容后才使用这些数据。通常“足够”的意思是指数据流已经结束，当然也可能是其他条件。

例 4-17　使用缓冲池模型来读取完整的流数据

```
          //stream 是个抽象的数据流
var spool = "";
stream.on('data', function(data) {
  spool += data;
});
stream.on('end', function() {
  console.log(spool);
});
```

4.3.2　文件系统

文件系统模块显然非常有用，因为你需要它来访问磁盘上的文件。它几乎模仿了文件 I/O 的 POSIX 风格。这个独特的模块为它所有的功能都提供了异步和同步的方法。但是，我们强烈建议你使用异步的方法，除非你是用 Node 来创建命令行脚本。即使这样，通常使用异步版本也会更好，虽然会增加一点点代码，但你可以并行访问多个文件，并缩减脚本运行的时间。

人们在处理异步调用时遇到的主要问题是执行次序具有不确定性，特别是处理文件 I/O 时。人们常常会想要同时进行一些操作，如文件移动、重命名、复制、读写等，但是其中一些操作依赖于另一些操作，所以当执行完成的次序不能确定时，可能会出现问题。这意味着代码中先执行的操作，有可能会在第二个操作之后才完成。有些模式能够很容易地解决次序问题，我们在第 3 章已经详细讨论过，在这里再重温一下。

考虑一下例 4-18 中的情况，读取文件然后把它删除掉。如果删除（unlink）发生在读取之前，就不可能读取到文件的内容了。

例 4-18　异步读取并删除文件——但这是错误的

```
var fs = require('fs');
```

```
fs.readFile('warandpeace.txt', function(e, data) {
  console.log('War and Peace: ' + data);
});

fs.unlink('warandpeace.txt');
```

需要注意的是，我们使用了异步方法，并且还创建了回调函数，但是并没有写任何代码来指定这些函数调用的次序。这对于不熟悉使用事件循环来编程的程序员来说，通常会导致一些问题。这个代码表面上看起来没问题，但运行起来有时候能正常工作，有时候却不能。需要使用一种模式，让我们可以指定想要运行的次序。有几种方法可实现这一点，其中一种常用的方法是采用回调函数嵌套。在例 4-19 中，删除文件的异步调用是嵌入在异步读取文件的回调函数中的。

例 4-19　通过嵌入回调函数完成异步读取并删除文件

```
var fs = require('fs');

fs.readFile('warandpeace.txt', function(e, data) {
  console.log('War and Peace: ' + data);
  fs.unlink('warandpeace.txt');
});
```

这个方法通常能够有效地把一组操作分离开。我们的例子中只有两个操作，很容易就能读懂理解。但这个模式有时候也可能会失去控制。

4.3.3　Buffer

虽然 Node 也是使用 JavaScript，但它是在 JavaScript 通常使用的环境外运行的。比如，浏览器需要 JavaScript 来进行许多操作，但并不包括处理二进制数据。虽然说 JavaScript 支持字节位操作，但它并没有二进制数据的原生表现形式。当你考虑到 JavaScript 里数字类型系统的限制时，更是会头疼不已，最后会变成只好采用二进制形式。Node 带来了 `Buffer` 类，为你操作二进制数据弥补了短板。

`Buffer` 是 V8 引擎上的扩展，这意味着它有其固有的一些限制。`Buffer` 实际上是对内存的直接分配，这意味着这多少受制于你在低级计算机语言方面的经验。JavaScript 的其他数据类型都把存储数据的复杂性进行了抽象，而 `Buffer` 与它们不同，它提供的是内存的直接操作。创建了一个 `Buffer` 后，它的大小就固定了。如果你需要添加更多的数据，就必须把老的 `Buffer` 复制到一个更大的 `Buffer` 中。虽然有些特性看起来让人沮丧，但它们让 `Buffer` 能够在服务器上快速地处理大量的数据操作。这是一个特意的设计选择，为了性能而牺牲了一些程序员的开发便利。

1. 二进制的快速入门

我们觉得有必要在这里插入使用二进制数据的快速入门内容，一方面是为了那些并没有太多二进制数据处理经验的读者，另一方面，对于那些很久没处理二进制数据的读者来说，正好也可以回顾一下（就像我们开始使用 Node 时的情况一样）。大多数人都知道，计算机的工作原理是操作“开”和“关”状态。因为只有这样两种状态，所以我们称此为*二元状态*。计算机的所有东西都建立在此基础上，这就说明了为什么在计算机上操作时，直接操作二进制通常是最快的方法。要做更复杂的事情时，我们把比特（bit，每一位表示一个二元状态）集合成 8 个一组，称之为 8 位字节（octet），也就是通常所说的字节（byte）。[3] 这样我们就能表示除了 0、1 外的其他数字了。

利用 8 位字节，我们就可以表示从 0 到 255 间的所有数字了。最右边的位表示 1，然后每向左移动一位，把它的表示值乘以 2。要计算某个字节表示的数字，只要简单地把对应位置上的数加起来就知道了（例 4-20）。

例 4-20　在一个字节里表示 0 到 255

```
128 64 32 16 8 4 2 1
--- -- -- -- - - - -
0   0  0  0  0 0 0 0 = 0

128 64 32 16 8 4 2 1
--- -- -- -- - - - -
1   1  1  1  1 1 1 1 = 255

128 64 32 16 8 4 2 1
--- -- -- -- - - - -
1   0  0  1  0 1 0 1 = 149
```

你还将接触到许多采用十六进制表示法（hex）的地方。因为字节需要用简单的方式来描述，但 8 个 0 和 1 组成的字符串并不够方便，所以十六进制表示法便流行起来。二进制表示法的基数是 2，因此每个数字（0 或 1）只有两种状态。十六进制使用的基数是 16，每一位能够表示 0 到 F 种状态，其中字母 A 到 F（或对应的小写字母）对应代表 10 到 15。十六进制非常方便的地方在于，我们只需要 2 个字母就能表示整个字节了。最右的位表示 1，往左一位就表示 16。如果我们要表示数字 149，就等价于（16×9）+（5×1），也就是十六进制的 95。

例 4-21　用十六进制表示 0 到 255

```
十六进制转换到十进制 :
```

注 3：并没有“标准”的字节大小，但实际上现在人们使用的通常是 8 位字节。因此 8 位字节（octet）和字节（byte）是等价的，我们会采用更常见的术语字节（byte）来指代特定的 8 位字节。

```
0 1 2 3 4 5 6 7 8 9 A  B  C  D  E  F
- - - - - - - - - - -- -- -- -- -- --
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

十六进制法计数：

```
16 1
-- -
0  0 = 0

16 1
-- -
F  F = 255

16 1
-- -
9  5 = 149
```

在 JavaScript 中，用一个十六进制值表示数字时，需要在其十六进制数值前添加 `0x` 标记。比如，`0x95` 表示十进制数字 149。在 Node 里，你会常常看见 `console.log()` 输出，或是在 Node 命令行中，`Buffer` 值是采用十六进制表示的。例 4-22 演示了如何用 `Buffer` 保存 3 个字符（比如 RGB 颜色值）。

例 4-22　用 8 位字节数组创建一个 3 个字节的 Buffer

```
> new Buffer([255,0,149]);
<Buffer ff 00 95>
>
```

那么二进制如何表示其他类型的数据呢？前面我们已经看见如何用二进制来表示数字了。在网络协议中，通常会指定一些字符来传达信息，比如用固定位置上的比特来表示特殊的含义。举个例子，在 DNS 请求中，头两个字节表示的数字是事务 ID，下一个字节的每个比特都是独立使用的，每一位表示了在这个请求中是否使用 DNS 的某个功能。

二进制另外一种非常常用的情况是用来表示字符串。其中使用最多的字符串编码方式是 ASCII 和 UTF（通常是 UTF-8）。这些编码方式定义了如何把比特转换成字符。我们不会在此深入地展开讨论，但基本上，编码的工作原理就是采用一个查找表把字符映射到对应的数字上。要把编码后的内容转换回来，计算机只要通过查找转换表，就能把数字变成字符。

ASCII 字符（其中包含非可见字符，如回车）一定是正好每个都是 7 位大小的，因此能表示 0 到 127 之间的数值。字符中的第 8 位比特通常用来扩展字符集，表示各种国际化的字符（如 ȳ 或 ō）。

UTF 则要复杂一些。它的字符集包含了非常多的字符，包括许多国际化字符。每个

UTF-8 字符需要至少 1 个字节，最多的时候需要 4 个字节才能表示。实质上，头 128 个值是传统的 ASCII，其他的值被推到了映射表的更远的地方，通过更大的数字来表示。当一个罕见的字符被引用时，通过第一个字节表示的数字，会告诉计算机利用下一个字节从映射表的第二页中查找字符的实际址。如果该字符不在映射表的第二页，第二个字节会告诉计算机去查找第三页，以此类推。这意味着在 UTF-8 中，字符串对应的长度并不是与字节数的长度一样。而 ASCII 中，这两个长度是永远一致的。

2. 二进制与字符串

需要重点记住的是，一旦你把内容复制到一个 `Buffer` 后，它就会以二进制的形式存储起来。当然，你可以随时把 `Buffer` 中的二进制内容转换成其他形式，比如说字符串。所以 `Buffer` 只由它的大小来定义，而非通过编码或者其他任何指示含义的方式。

既然 `Buffer` 是不透明的，那么它需要多大才能把输入的特定字符串保存起来呢？正如前面我们介绍的，一个 UTF 字符可能会占用最多 4 个字节。因此为安全起见，需要定义一个 `Buffer` 的大小为可能输入的 UFT 字符最大值的 4 倍大小。你可以采取一些方法来减轻这个负担，比如限制输入只能为欧洲语言，这样就能确定每个字符最大为 2 个字节了。

3. `Buffer` 的使用

创建 `Buffer` 可以使用 3 种参数：指定 `Buffer` 的字节长度，需要拷贝到 `Buffer` 里的字节数组，或是需要拷贝到 `Buffer` 里的字符串。第一和最后一种方法是目前最常使用的。在一些不常见的情况下，你会需要用一个 JavaScript 的字节数组。[4]

创建特定大小的 `Buffer` 是很常见的情况，而且容易处理。你只需在创建 `Buffer` 的时候指定需要的字节大小作为参数就可以了（例 4-23）。

例 4-23　指定字节长度创建 `Buffer`

```
> new Buffer(10);
<Buffer e1 43 17 05 01 00 00 00 41 90>
>
```

正如你在前一个例子中所见，创建 `Buffer` 后，得到了一个对应长度的字节组。但是，因为 `Buffer` 是从内存直接分配的，它并不会对原有的内容进行初始化，所以得到的内容就是原本占用的东西。这与原生的 JavaScript 类型不同，它们会把所有的内存初始化，无论你是创建一个新的原生变量还是对象，它都不会把原本内存空

注 4：其中一个原因是这样非常浪费内存。比如，若把每个字节作为一个数字保存，你需要使用 64 位大小的内存空间来表示一个 8 位大小的内容。

间的垃圾数据返回给你。你可以用下面的情景来帮助理解。假设你到了一家繁忙的咖啡店，想找一张桌子时，最快的方法就是一旦有人离开了就立马坐下来。虽然这样很快，但是你会面对之前客人留下的脏盘子和剩菜。你也许希望等待服务员清理桌子后再坐下。这与 `Buffer` 和原生类型的工作方式很像。`Buffer` 并不会为了让你更方便而做额外的工作，但它们能让你直接快速地操作内存。如果你想要一组漂亮的全是 0 的比特组，就需要自己动手（或是找找其他工具库）。

当你在处理网络传输协议之类的工作时，因为它们有着定义好的格式，所以创建指定字节长度的 `Buffer` 就很常用了。当你准确地知道数据的大小（或者是知道最大会是多少），并为了性能原因想分配并重用 `Buffer` 时，这就是很好的选择。

也许创建 `Buffer` 最常用的方法就是使用 ASCII 或 UTF-8 字符串了。虽然 `Buffer` 可以存储任何数据，但在处理 I/O 的字符数据时 `Buffer` 特别有用，因为 `Buffer` 本身的一些限制使得它的操作比一般的字符串操作要快很多。所以当你在创建高度可扩展的应用时，通常值得采用 `Buffer` 来保存字符串，特别是当你只是在应用间分流字符串，而不会修改它们的时候。因此，即使 JavaScript 原生存在了字符串类型，在 Node 程序中还是会经常使用 `Buffer` 来保存字符串。

如例 4-24 所示的例子，我们用字符串来创建 `Buffer`，它默认是 UTF-8 编码的。如果你没有指定编码格式，它就会认为是 UTF-8 字符串。这并不意味着 `Buffer` 会把字符串补全成能够存下任意 Unicode 字符的大小（盲目地为每个字符分配 4 个字节），而是说明它不会截断字符内容。在这个例子中，我们看到当输入的字符串是小写字母时，无论采用的是哪种编码方式，`Buffer` 都使用同样的字节结构，因为每个字母都落在同样的区间里。但是，当我们输入“é”字符时，无论是默认的 UTF-8 还是我们显式指定为 UTF-8，它都被编码成 2 个字节大小。但是当我们指定编码为 ASCII 时，字符被截断成单个字节。

例 4-24　用字符串创建 `Buffer`

```
> new Buffer('foobarbaz');
<Buffer 66 6f 6f 62 61 72 62 61 7a>
> new Buffer('foobarbaz', 'ascii');
<Buffer 66 6f 6f 62 61 72 62 61 7a>
> new Buffer('foobarbaz', 'utf8');
<Buffer 66 6f 6f 62 61 72 62 61 7a>
> new Buffer('é');
<Buffer c3 a9>
> new Buffer('é', 'utf8');
<Buffer c3 a9>
> new Buffer('é', 'ascii');
<Buffer e9>
>
```

4. 字符串的使用

Node 提供了一些操作来简化字符串和 Buffer 操作。首先，你不需要在创建 Buffer 前提前计算字符串的长度，只要把字符串作为参数传给创建 Buffer 的函数就可以了。或者，你也可以使用 Buffer.byteLength() 方法来获得字符串在编码上的字节长度，而不是 String.length 返回的字符个数。

你还可以往已经存在的 Buffer 上写入字符串。Buffer.write() 会把字符串写到 Buffer 指定的位置上。如果从 Buffer 指定位置开始有足够空间的话，整个字符串都会被写入。否则，字符串的尾部会被截断，好让其大小能放入 Buffer。在这两种情况下，Buffer.write() 都会返回一个数字，表示有多少字节被成功写入。对于 UTF-8 字符串来说，如果一个完整字符无法写入到 Buffer 的话，就不会单独写入该字符的某个字节。如在例 4-25 中，因为 Buffer 太小了，以至于无法写入一个非 ASCII 字符，所以它就是空的。

例 4-25　Buffer.write() 及部分字符

```
> var b = new Buffer(1);
> b
<Buffer 00>
> b.write('a');
1
> b
<Buffer 61>
> b.write('é');
0
> b
<Buffer 61>
>
```

在只有一个字节的 Buffer 中，它可以写入一个“a”字符，所以操作返回了 1，表示写入了 1 个字节。但是，尝试写入一个“é”字符的时候，它需要 2 个字节，因此操作返回的是 0，因为没有写入任何东西。

使用 Buffer.write() 还有些复杂的情况。如果条件允许，当写入 UTF-8 时，Buffer.write() 写入的字符串会以一个 NULL 字符结尾[5]。这在一个较大的 Buffer 中写入时能够看得更明显。

在例 4-26 中，创建了一个 5 个字节长的 Buffer（这是通过传入字符串直接完成的）。我们用字母 f 把 Buffer 整个写满，f 的字符编码是 0x66（十进制是 102）。这是为了让我们能够看清楚，在 Buffer 位移为 1 的地方写入字符“ab”会有什么效果。第 0 位的字符依然是 f。在位置 1 和 2 上，字符被改为了 61 和 62。然后

注 5：通常这只是意味着是一个二进制的 0。

Buffer.write() 插入了一个结束符，正如例子中的一个空字符 0x00。

例 4-26　写入 Buffer 的字符串包含了结束符

```
> var b = new Buffer(5);
> b.write('fffff');
5
> b
<Buffer 66 66 66 66 66>
> b.write('ab', 1);
2
> b
<Buffer 66 61 62 00 66>
>
```

4.3.4　console.log

这个简单的 console.log 命令借用了 Firefox 中 Firebug 调试器的概念，让你可以轻松把输出打印到标准输出（stdout），而不需要借助任何模块（例 4-27）。它还提供了美化打印格式的功能来帮助遍历对象。

例 4-27　用 console.log 输出

```
> foo = {};
{}
> foo.bar = function() {1+1};
[Function]
> console.log(foo);
{ bar: [Function] }
>
```

第 5 章

工具类API

本章会介绍另一组你肯定会经常使用的 API，但它们的使用频率不如第 4 章介绍的那些 API 高。

5.1 DNS

和普通用户一样，程序员一般都希望用域名来代替 IP 地址作为事物的引用名称。DNS 模块就提供了这种查找的功能，也为那些使用域名的模块提供支持，如 `HTTP` 客户端。

DNS 模块包含了两个主要方法，以及一些便利的方法。这两个主要方法是 `resolve()` 和 `reverse()`，前者把域名转换成 DNS 记录，后者将 IP 地址转换成域名。DNS 模块的其他方法都是这两种方法的特殊形式。

`dns.resolve()` 接受以下 3 个参数。

待解析的域名字符串

这可以包含子域名，如 www.yahoo.com。其中 www 是主机名，但系统也会为你解析它。

表示请求的记录类型的字符串

这需要你对 DNS 有更多的了解。许多人都熟悉 address 或 A record 类型，这种

记录类型把 IPv4 区域映射到一个域名（前一个项目定义的）。而 canonical name 或 CNAME 记录允许你为 A record 或另外一个 CNAME 创建一个别名，如 www.example.com 可能是 A record 类型域名 example.com 的别名。MX 记录指向使用 SMTP 的邮件域名服务器。当你发送 email 到 person@domain.com 时，domain.com 的 MX 记录会告诉你的邮件服务器该把邮件发往哪里。Text 记录，或称为 TXT，是依附在域名上的记录，它可以用作各种用途。DNS 库支持的最后一种类型是 service，或称为 SRV 记录，它的作用是在特定域名下说明有哪些服务可用。

回调函数

这是从 DNS 服务器返回的响应内容。函数原型见例 5-2。

如例 5-1 所示，调用 `dns.resolve()` 很简单，但它的回调函数与你之前看见的其他回调函数有些不一样。

例 5-1　调用 `dns.resolve()`

```
dns.resolve('yahoo.com', 'A', function(e,r) {
  if (e) {
    console.log(e);
  }
  console.log(r);
} );
```

我们调用 `dns.reslove()` 时指定了域名和记录类型 `A`，同时还有一个简略的回调函数把结果打印出来。回调函数的第一个参数是 `error` 对象。如果有错误发生，该对象就不会是 `null`，我们就能通过它查看到底发生了什么错误。第二个参数是查询返回的结果列表。

还有些方便的方法可用来处理前面列出的各种记录类型。比如，除了调用 `resolve('example.com', 'MX', callback)` 以外，你还可以调用 `resolveMx('example.com', callback)`，见例 5-2。API 还提供了 `resolve4()` 和 `resolve6()` 方法，分别用来解析 IPv4 和 IPv6 地址。

例 5-2　使用 `resolve()` 和 `resolveMx()`

```
var dns = require('dns');

dns.resolve('example.com', 'MX', function(e, r) {
  if (e) {
    console.log(e);
  }
  console.log(r);
});

dns.resolveMx('example.com', function(e, r) {
```

```
    if (e) {
      console.log(e);
    }
    console.log(r);
  });
```

因为 resolve() 通常会返回一个包含许多 IP 地址的列表，所以需要有 dns 模块，该模块提供了一个便利的 dns.lookup() 方法，可以从一个 A 记录查询中只返回一个 IP 地址（例 5-3）。该方法的参数是域名、IP 类型（4 或 6）和回调函数。但是，与 dns.resovle() 不同，它永远只返回一个地址。如果你没有传入地址，它会默认是网络设备接口的当前设置。

例 5-3　用 lookup() 查询单个 A 记录

```
var dns = require('dns');

dns.lookup('google.com', 4, function(e, a) {
  console.log(a);
});
```

5.2　加密

加密在许多领域都会用到，Node 的加密算法是以 OpenSSL 库为基础的，这是因为 OpenSSL 的加密算法经过了充分测试，并且有着良好的实现。但你需要在编译 Node 的时候指定添加 OpenSSL 支持，才能使用本节介绍的方法。

加密模块能帮你完成以下工作。首先，它使 Node 能够使用 SSL/TLS。其次，它包含的哈希算法，如 MD5 或 SHA-1，也许是你在开发应用中会用到的。再次，它允许你使用 HMAC[1]，其提供了若干加密方法来确保数据安全。最后，HMAC 包含了公钥加密功能来对数据进行签名及验证。

加密的每个功能都包含在一个或多个类中，我们接下来会一一介绍。

5.2.1　Hashing

哈希常用于几个重要的功能，比如把数据混淆以便于验证，或是为一个较大的数据提供很小的校验。要在 Node 里使用哈希，需要调用工厂方法 crypto.createHash() 来创建一个 Hash 对象。它会返回指定哈希算法的 Hash 新实例，大部分流行的算法都包含在内，具体支持哪种算法要看你安装的 OpenSSL 的版本，几个常见的算法有：

注 1：Hash-based Message Authentication Code（HMAC）是一种验证数据的加密算法。它通常被用来验证两个数据是否一致，其作用类似哈希算法，但也可以用来确保数据没有被篡改。

- md5
- sha1
- sha256
- sha512
- ripemd160

这些算法有各自的优缺点。比如 MD5 在许多应用里都会用到，但它有一些已知的缺陷，包括碰撞问题[2]。根据实际的应用需要，你可以选择广泛使用的算法（如 MD5），或者更新的 SHA1（推荐使用），甚至是更少见但更健壮的算法（如 RIPEMD、SHA256 或 SHA512）。

在哈希中使用数据时，可以调用 `hash.update()` 来生成数据摘要（digest，见例 5-4）。你可以用更多的数据不停地更新哈希，直到需要把它输出为止。你添加到哈希对象的数据只是简单地追加到前一次传入的数据尾部。要把哈希输出，只需调用 `hash.digest()` 方法，这会把所有通过 `hash.update()` 输入的数据生成摘要并输出。在调用 `hash.digest()` 之后，就不可以再添加任何输入进去了。

例 5-4　用 Hash 创建摘要

```
> var crypto = require('crypto');
> var md5 = crypto.createHash('md5');
> md5.update('foo');
{}
> md5.digest();
'-½\u0018ÛLÂø\\íïeOÌÄ¤Ø'
>
```

注意，输出的摘要看起来有点诡异，这是因为它是以二进制的格式呈现的。通常，摘要是用十六进制打印的，我们可以在调用 `hash.digest` 的时候传入 `'hex'` 作为参数，如例 5-5 所示。

例 5-5　哈希的生命周期和得到十六进制输出

```
> var md5 = crypto.createHash('md5');
> md5.update('foo');
{}
> md5.digest();
'-½\u0018ÛLÂø\\íïeOÌÄ¤Ø'
> md5.digest('hex');
Error: Not initialized
    at [object Context]:1:5
    at Interface.<anonymous> (repl.js:147:22)
    at Interface.emit (events.js:42:17)
```

注 2：要想故意找出两个数据使得它们的 MD5 校验一样也是可能的，这就使得在某些情况下这个方法不太合适。一些更现代的算法更加安全，虽然现在有人也在 SHA1 上发现了类似的问题。

```
    at Interface._onLine (readline.js:132:10)
    at Interface._line (readline.js:387:8)
    at Interface._ttyWrite (readline.js:564:14)
    at ReadStream.<anonymous> (readline.js:52:12)
    at ReadStream.emit (events.js:59:20)
    at ReadStream._emitKey (tty_posix.js:280:10)
    at ReadStream.onData (tty_posix.js:43:12)
> var md5 = crypto.createHash('md5');
> md5.update('foo');
{}
> md5.digest('hex');
'acbd18db4cc2f85cedef654fccc4a4d8'
>
```

当再次调用 `hash.digest()` 时，我们得到了一个错误。这是因为一旦调用 `hash.digest()`，`Hash` 对象就已经最终确定，而且不能被重用。我们需要创建一个 `Hash` 的新实例来使用，这次得到的十六进制输出更为有用。`hash.digest()` 输入的选项包括二进制（默认）、十六进制和 base64 编码。

因为数据在 `hash.update()` 调用时是串联起来的，所以例 5-6 中两个示例得到的结果是一样的。

例 5-6　看看 `hash.update()` 是如何把输入连接起来的

```
> var sha1 = crypto.createHash('sha1');
> sha1.update('foo');
{}
> sha1.update('bar');
{}
> sha1.digest('hex');
'8843d7f92416211de9ebb963ff4ce28125932878'
> var sha1 = crypto.createHash('sha1');
> sha1.update('foobar');
{}
> sha1.digest('hex');
'8843d7f92416211de9ebb963ff4ce28125932878'
>
```

虽然 `hash.update()` 看起来和流模式很像，但要注意的是，其实它并不是流。你可以很容易地把一个流连接到 `hash.update()` 上，但是不能使用 `stream.pipe()` 方法。

5.2.2　HMAC

HMAC 结合了哈希算法和加密密钥，是为了阻止对签名完整性的一些恶意攻击。这意味着 HMAC 同时使用了哈希算法（如上一小节所介绍的一些算法）以及一个加密密钥。Node 提供的 HMAC API 和 Hash API 是一样的。唯一的不同是，创建 `hmac` 对象时需要在传入哈希算法的同时，再传入一个密钥。

crypto.createHmac() 返回的是一个 Hmac 的实例，提供了 update() 和 digest() 方法，这些和之前我们介绍的 Hash 方法一模一样。

创建 Hmac 对象需要的密钥必须是一个 PEM 编码的密钥，以字符串的格式传入。如例 5-7 所示，在命令行用 OpenSSL 可以轻松创建一个密钥。

例 5-7　创建 PEM 编码的密钥

```
Enki:~ $ openssl genrsa -out key.pem 1024
Generating RSA private key, 1024 bit long modulus
...++++++
............................++++++
e is 65537 (0x10001)
Enki:~ $
```

这个例子创建的是一个 PEM 格式的 RSA 密钥，并保存在一个文件里（在本例中是 key.pem）。如果我们忽略 -out key.pem 参数，也可以在 Node 中使用 process 模块来直接调用同样的功能（本章后面会介绍到）。用这个方法，我们将在标准输出中得到结果，否则，就需要从文件中读取密钥，然后再用来创建 Hmac 对象并生成摘要了（例 5-8）。

例 5-8　创建 Hmac 摘要

```
> var crypto = require('crypto');
> var fs = require('fs');
>
> var pem = fs.readFileSync('key.pem');
> var key = pem.toString('ascii');
>
> var hmac = crypto.createHmac('sha1', key);
>
> hmac.update('foo');
{}
> hmac.digest('hex');
'7b058f2f33ca28da3ff3c6506c978825718c7d42'
>
```

这个例子用到了 fs.readFileSync()，因为在许多情况下，读取密钥会放在服务器启动任务中。这个时候，我们可以用同步的方式来读取密钥（但会使服务器启动时间稍微变长）。因为你还没开始服务任何客户，所以把事件循环堵塞一会儿并没有什么问题。一般情况下，除了使用加密密钥外，使用 Hmac 与使用 Hash 的例子是一样的。

5.2.3　公钥加密

公钥加密功能分布在如下 4 个类中：Cipher、Decipher、Sign 和 Verify。和加密模块的其他类一样，它们也有工厂方法。Cipher 把数据加密，Decipher 解密数

据，`Sign` 为数据创建加密签名，`Verify` 验证加密签名。

对 HMAC 操作，我们用到了私钥。对于本小节的操作，我们将同时使用公钥和私钥。公钥加密算法需要一组配对的密钥：一个是私钥，由物主保存，用来解密和对数据签名，另外一个是公钥，提供给第三方。公钥可以用来加密数据，并且只能让私钥拥有者解读，或者用来验证数据是否被对应的私钥所签名。

让我们从刚刚生成来进行 HMAC 摘要的私钥中提取对应的公钥吧（例 5-9）。Node 要求公钥按照证书格式，所以需要你提供额外的信息，但你也可以不填这些信息，让其留空就行了。

例 5-9　从私钥中提取公钥证书

```
Enki:~ $ openssl req -key key.pem -new -x509 -out cert.pem
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgets Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (eg, YOUR name) []:
Email Address []:
Enki:~ $ ls cert.pem
cert.pem
Enki:~ $
```

我们让 OpenSSL 读取私钥，然后把公钥以 X509 证书格式输出到 cert.pem 文件中。加密算法用到的密钥都要求是 PEM 格式的。

1. 用 Cipher 加密

`Cipher` 类提供了用私钥加密数据的功能。该工厂方法输入一个算法和私钥，然后创建 `cipher` 对象。支持的算法是从你安装的 OpenSSL 实现中支持的：

- `blowfish`
- `aes192`

许多现代的加密算法使用块密码，也就是输出的通常是标准大小的“块”。块大小与使用的算法有关，如 `blowfish` 使用的是 40 字节的块。这当你使用 `Cipher` API 时会明显看出，因为 API 总是使用固定大小的块。这种做法能够防止信息泄露给攻击

者，如加密的信息或者是用来加密的特定密钥。

和 Hash、Hmac 类似，Cipher API 也采用 update() 方法来输入数据，但是在使用 cipher 时 update() 的工作方式不一样。首先，如果条件允许，cipher.update() 会返回一块加密的数据。这时候数据块的大小变得很重要，如果 cipher 中的数据加上传给 cipher.update() 的数据足够用来创建一个或多个加密块，那么这些加密数据就会被返回。如果数据不足以构成一个加密块，输入会被保存在 cipher 对象内。Cipher 还有一个新的方法 cipher.final()，它会代替 digest() 方法。当调用 cipher.final() 时，cipher 对象中剩余的所有数据都会被加密并返回，但会添加足够的填充使其满足块大小的要求（例 5-10）。

例 5-10　密码与块大小

```
> var crypto = require('crypto');
> var fs = require('fs');
>
> var pem = fs.readFileSync('key.pem');
> var key = pem.toString('ascii');
>
> var cipher = crypto.createCipher('blowfish', key);
>
> cipher.update(new Buffer(4), 'binary', 'hex');
''
> cipher.update(new Buffer(4), 'binary', 'hex');
'ff57e5f742689c85'
> cipher.update(new Buffer(4), 'binary', 'hex');
''
> cipher.final('hex')
'96576b47fe130547'
>
```

为了让例子易于阅读，我们指定了输入输出格式。输入输出格式都是可选的，如果没有指定，将默认按二进制处理。这个例子中，我们指定了输入格式为二进制，因为传入了一个新的 Buffer 对象（包含了内容已有的垃圾数据）。同时还指定了输出格式为十六进制，这是为了让产生的内容更容易读。你可以看到，第一次调用 cipher.update() 时传入了 4 个字节的数据，得到的是一个空字符串；第二次，因为有足够的数据来生成加密块，我们得到了十六进制格式的加密数据。当调用 cipher.final() 时，因为数据并不够用来创建一个完整的加密块，所以输出被填补成一个完整（最终）块并返回。如果我们发送的数据超过一个块所需要的大小，cipher.final() 会先返回尽可能多的加密块，然后才会采用补全的方法。因为 cipher.final() 只是用来把已有的数据输出，所以它并不接受新的输入内容。

2. 用 Decipher 解密

Decipher 类几乎就是 Cipher 类的反面。你可以把加密的数据通过 decipher.

update()传给一个Decipher对象，它会把数据以流的形式保存成块，并在数据足够的时候输出解密数据。你也许会想，因为cipher.update()和cipher.final()总是输出固定大小的数据块，你也不得不给Decipher精确大小的内容。但幸运的是，它会缓存数据。而且，你还可以用其他I/O传输方式来传给它数据，如磁盘和网络，即使这些方法给你的数据大小与加密算法使用的不一样。

让我们看一下例5-11，本例演示了如何加密数据并解密它。

例 5-11　文本加密与解密

```
> var crypto = require('crypto');
> var fs = require('fs');
>
> var pem = fs.readFileSync('key.pem');
> var key = pem.toString('ascii');
>
> var plaintext = new Buffer('abcdefghijklmnopqrstuv');
> var encrypted = "";
> var cipher = crypto.createCipher('blowfish', key);
> ..
> encrypted += cipher.update(plaintext, 'binary', 'hex');
> encrypted += cipher.final('hex');
>
> var decrypted = "";
> var decipher = crypto.createDecipher('blowfish', key);
> decrypted += decipher.update(encrypted, 'hex', 'binary');
> decrypted += decipher.final('binary');
>
> var output = new Buffer(decrypted);
>
> output
<Buffer 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f 70 71 72 73 74 75 76>
> plaintext
<Buffer 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f 70 71 72 73 74 75 76>
>
```

重要的是要确保输入输出的格式在纯文本和加密数据上是一致的。还要注意的是，为了得到一个Buffer对象，你需要用Cipher和Decipher返回的字符串来构建。

3. 用 Sign 来创建签名

Signatures验证的是签名者是否用其私钥对数据进行授权。但是，和HMAC不同，公钥可以用来对签名进行认证。Sign类的API与HMAC的几乎一样（见例5-12）。crypto.createSign()用来创建sign对象；createSign()只需要传入签名算法；sign.update()可给sign对象添加数据。想创建签名时，可以用你的私钥来调用sign.sign()给数据进行签名。

例 5-12　用 sign 对数据进行签名

```
> var sign = crypto.createSign('RSA-SHA256');
> sign.update('abcdef');
{}
> sig = sign.sign(key, 'hex');
'35eb47af5260a00c7bad26edfbe7732a897a3a03290963e3d17f48331a42...aa81b'
>
```

4. 用 Verify 来验证签名

Verify API 使用的方法和我们刚刚讨论的类似（见例 5-13），它用 verify.update() 来添加数据，且当你把需要验证的数据都添加好后，就可以调用 verify.verify() 对签名进行验证了，它需要传入证书（公钥）、签名以及签名的格式。

例 5-13　验证签名

```
> var crypto = require('crypto');
> var fs = require('fs');
>
> var privatePem = fs.readFileSync('key.pem');
> var publicPem = fs.readFileSync('cert.pem');
> var key = privatePem.toString();
> var pubkey = publicPem.toString();
>
> var data = "abcdef"
>
> var sign = crypto.createSign('RSA-SHA256');
> sign.update(data);
{}
> var sig = sign.sign(key, 'hex');
>
> var verify = crypto.createVerify('RSA-SHA256');
> verify.update(data);
{}
> verify.verify(pubkey, sig, 'hex');
1
```

5.3　进程

虽然 Node 把许多东西从操作系统中抽象出来，但你依然在操作系统里运行，而且可能想要更直接地与它交互。Node 中可以使用系统中已经存在的进程，或者创建新的子进程来做各种工作。虽然 Node 本身是一个“胖”线程，带有单独一个事件循环，但你可以任意地开启其他进程（线程）在事件循环外工作。

5.3.1 process模块

可以使用 process 模块从当前的 Node 进程中获得信息，并可以修改配置。和其他大部分模块不同，process 模块是全局的，并且可以一直通过变量 process 获得。

1. process 事件

process 是 EventEmitter 的实例，所以它提供了基于对 Node 进程的系统调用的事件。exit 事件提供了在 Node 进程退出前的最终响应时机（例 5-14）。重要的是，事件循环在 exit 事件之后就不会再运行了，因此只有那些不需要回调函数的代码才会被执行。

例 5-14　在 Node 退出前调用代码

```
process.on('exit', function () {
  setTimeout(function () {
   console.log('This will not run');
  }, 100);
  console.log('Bye.');
});
```

因为事件循环不会再运行，因此 setTimeout() 里的代码永远不会执行。

process 提供的一个非常有用的事件是 uncaughtException（例 5-15）。用过 Node 一段时间后，你会发现那些在事件主循环里碰到的异常会导致 Node 进程退出。在许多应用场景下，特别是对那些希望永不当机的服务器程序来说，这都是不可接受的。uncaughtException 事件会提供一个极其暴力的方法来捕获这些异常。它确实是最后一道防线了，但对解决此问题非常有效。

例 5-15　通过 uncaughtException 事件捕获异常

```
process.on('uncaughtException', function (err) {
  console.log('Caught exception: ' + err);
});

setTimeout(function () {
  console.log('This will still run.');
}, 500);

// 故意导致异常，并且不捕获它。
nonexistentFunc();
console.log('This will not run.');
```

让我们来分解一下整个操作过程。首先，我们为 uncaughtException 创建了一个事件监听器。它并非一个智能处理程序，只是简单地把异常输出到标准输出。如果这个 Node 脚本是作为服务器运行的话，可以很方便地把标准输出保存到一个文件中，记录下这些错误。但是，因为它捕获的是一个不存在的函数触发的事件，

所以虽然 Node 程序不会退出，但是标准的执行流程会被打断。我们知道所有的 JavaScript 都会执行一遍，然后任何回调函数都可能在其对应监听的事件触发时被调用到。但在这个例子中，因为 `nonexistentFunc()` 抛出了异常，所以在它之后的代码都不会执行下去。然而，在此之前已经运行的代码会继续下去，也就是说，`setTimeout()` 依然会被调用。这在你开发服务器程序的时候很重要。让我们再多看些这方面的例子（例 5-16）。

例 5-16　捕获异常的回调函数的作用

```
var http = require('http');
var server = http.createServer(function(req,res) {
  res.writeHead(200, {});
  res.end('response');
  badLoggingCall('sent response');
  console.log('sent response');
});

process.on('uncaughtException', function(e) {
  console.log(e);
});

server.listen(8080);
```

这段代码创建了一个简单的 HTTP 服务器，然后在进程层次上监听所有的未捕获异常。在 HTTP 服务器中，回调函数在发送了 HTTP 响应后，故意调用了一个错误的函数。例 5-17 展示了这个脚本的终端输出内容。

例 5-17　例 6-16 的输出

```
Enki:~ $ node ex-test.js
{ stack: [Getter/Setter],
  arguments: [ 'badLoggingCall' ],
  type: 'not_defined',
  message: [Getter/Setter] }
{ stack: [Getter/Setter],
  arguments: [ 'badLoggingCall' ],
  type: 'not_defined',
  message: [Getter/Setter] }
{ stack: [Getter/Setter],
  arguments: [ 'badLoggingCall' ],
  type: 'not_defined',
  message: [Getter/Setter] }
{ stack: [Getter/Setter],
  arguments: [ 'badLoggingCall' ],
  type: 'not_defined',
  message: [Getter/Setter] }
```

当启动这个例子的脚本时，服务器已经准备就绪，然后我们对它发起了几次 HTTP 请求。注意，服务器并没有关闭退出。相反，错误被绑定在 `uncaughtException`

事件上的函数记录了下来，而且我们依然能够提供完整的 HTTP 请求处理服务。这是为什么呢？Node 故意阻止了正在运行的回调函数继续处理和调用下面的 console.log()。该错误只会影响我们派生出来的进程，而服务器能够继续运行，所以该异常被封装在一个特定的代码路径上，其他代码并不会受影响。

还要重点理解的是 Node 中的监听器是如何实现的，让我们看看例 5-18。

例 5-18　EventEmitter 的简略监听代码

```
EventEmitter.prototype.emit = function(type) {

...

  var handler = this._events[type];

...

  } else if (isArray(handler)) {
    var args = Array.prototype.slice.call(arguments, 1);

    var listeners = handler.slice();
    for (var i = 0, l = listeners.length; i < l; i++) {
      listeners[i].apply(this, args);
    }
    return true;

...

};
```

在事件触发后，运行时处理程序中的一项检查是看看是否存在事件监听器的数组。如果有几个监听器，运行执行器会按数组顺序把里面的监听器一一调用。意思是说，第一个绑定的监听器会首先用 apply() 方法调用，然后是第二个，以此类推。这里需要重点注意的是，同一个事件的所有监听器是在同一个代码路径上的。所以如果其中一个回调函数出现了异常未被捕获，将导致该事件的其他回调函数终止执行。但是一个事件实例中的未捕获异常不会影响其他事件。

我们还能利用 process 来访问一些系统事件。当进程得到一个信号的时候，它会通过 process 触发的事件通知 Node 程序。例如操作系统会产生许多 POSIX 系统事件，如 sigaction(2) 帮助文档中介绍的那些。最常见的有 SIGINT、中断信号量。通常，当用户对运行在终端的程序按下 Ctrl-C 的时候，SIGINT 就会发生。除非你通过 process 来处理信号事件，否则 Node 会采取默认方法进行处理。比如说 SIGINT 的情况，默认操作就是立刻杀死进程。你可以通过 process.on() 方法来修改这些默认行为，除了一些永远无法捕获的信号之外（例 5-19）。

例 5-19　捕捉 Node 进程的信号量

```
// 开始从标准输入读取内容，所以程序不会退出
process.stdin.resume();

process.on('SIGINT', function () {
  console.log('Got SIGINT.  Press Control-D to exit.');
});
```

为了确保 Node 程序不会主动退出，我们从标准输入读取内容（详见下面第三小节的内容），这样 Node 进程就会继续运行了。如果你在程序运行的时候按下 Ctrl-C，操作系统会发送 SIGINT 信号给 Node 程序，这会被 SIGINT 事件处理器所捕获。在本例中，我们采用把信息记录在终端的方式来代替原本的退出程序操作。

2. 与当前 Node 进程交互

`process` 包含了有关 Node 进程的许多元信息。当你希望在进程内管理 Node 运行环境时，这会很有用。这里面包含了关于 Node 进程的若干不可改变（只读）的信息，例如：

- `process.version`
- 包含了正在运行的 Node 的版本号：

- `process.installPrefix`
- 也包含了安装时指定的安装目录 (/usr/local、~/local 等)：

- `process.platform`
- 会列出正在运行的平台名称。输出内容会指明内核（`linux2`、`darwin` 等），而不是 Redhat ES3、Windows 7、OSX 10.7 这一类名称。

- `process.uptime()`
- 还会列出当前进程运行了多少秒。

此外，你还可以从 Node 进程得到或设置一些属性。当进程运行时，它是按某个特定的用户及用户组启动的。你可以调用 `process.getgid()`、`process.setgid()`、`process.getuid()` 和 `process.setuid()` 来获得或修改这些属性。这样做可以有效地确保 Node 程序运行在一个安全的环境中。还需要注意的是，`set` 方法除了可以接受用户名 / 用户组所对应的数字 ID 外，还可以直接使用用户组 / 用户名本身。但是，如果你传入的是用户组或用户名，该方法会采取堵塞的方式来把这个信息翻译成 ID，这样会花费些时间。

正在运行的 Node 实例的进程 ID，或称为 PID，可以通过 `process.pid` 属性得到。

你还能修改 process.title 属性来设置 Node 显示在系统的标题名称，该属性修改后的内容会在 ps 命令调用时显示出来。当你在生产环境中需要运行多个 Node 进程时，这会很有用。你可以为每个进程修改容易辨别的名称，而不是一堆进程都叫做 node（或者 node app.js）。当一个进程占用了大量的 CPU 或 RAM 时，也可以很快地知道具体是谁干的。

其他可用的信息包括 process.execPath，它显示的是当前执行的 node 程序所在的路径，比如 /usr/local/bin/node。当前的工作目录（所有打开文件的相对路径）可以用 process.cwd() 获取。工作目录是 Node 启动的目录。你可以调用 process.chdir() 来修改（如果修改的目录不可读或者不存在，将会抛出异常），还可以使用 process.memoryUsage() 来得到当前进程的内存使用情况，这会返回一个对象来说明内存使用的各种情况：rss 是 RAM 的使用量，而 vsize 是内存使用总量，包括了 RAM 和 swap。你还可以获知 V8 的一些状态：heapTotal 和 heapUsed 分别表示 V8 分配了多少内存，已经有多少内存正在使用。

3. 操作系统的输入 / 输出

通过 process，还有若干方法可以与操作系统交互（除了修改正在运行的 Node 进程以外）。其中一个主要功能就是可以访问操作系统的标准 I/O 流，stdin 是进程的默认输入流，stdout 是进程的输出流，stderr 是其错误输出流。它们对应暴露的接口是 process.stdin、process.stdout 和 process.stderr，其中 process.stdin 是可读的数据流，而 process.stdout 和 process.stderr 是可写的数据流。

(1) process.stdin　stdin 在进程间通信时是非常有用的，它能够为命令行下采用管道通信提供便利。当我们输入 cat file.txt | node program.js 时，标准输入流会接收到 cat 命令输出的数据。

因为任何时候都能使用 process，所以 process.stdin 也会为所有的 Node 进程初始化。但它一开始是处于暂停状态，这时候 Node 可以对它进行写入操作，但是你不能从它读取内容。在尝试从 stdin 读数据之前，需要先调用它的 resume() 方法（见例 5-20）。Node 会为此数据流填入供读取的缓存，并等待你的处理，这样可以避免数据丢失。

例 5-20　把标准输入写到标准输出

```
process.stdin.resume();
process.stdin.setEncoding('utf8');

process.stdin.on('data', function (chunk) {
  process.stdout.write('data: ' + chunk);
});
```

```
process.stdin.on('end', function () {
  process.stdout.write('end');
});
```

我们请求 process.stdin 进行 resume() 操作，并把编码设置为 UTF-8，然后设置了监听器把接收的数据推送到 process.stdout 上去。当 process.stdin 发起 end 事件时，我们把它也传输给 process.stdout 流。因为 stdin 和 stdout 都是真正的数据流，所以我们也可以采用更简便的方法，那就是使用数据流的 pipe() 方法，如例 5-21 所示。

例 5-21　通过管道把标准输入转到标准输出

```
process.stdin.resume();
process.stdin.pipe(process.stdout);
```

这是连接两个数据流的最漂亮的方式。

(2) process.stderr　stderr 用来输出异常和程序运行过程中遇到的问题。在 POSIX 系统里，因为它是另外一个独立的流，所以输出的日志和错误的日志很容易被记录到不同的目标位置。这也许是可取的，但 Node 有自己的一套处理特性。当写入 stderr 时，Node 将保证该次写入的会被完成。但是，和其他普通的流不一样，这会以堵塞的方式执行。通常情况下，调用 Stream.write() 会返回一个布尔值，用来表示 Node 是否能够写到内核缓存中去。对于 process.stderr 来说这个返回值永远是真，但它可能不会像一般的 write() 那样立刻返回，而是需要等待一会儿。一般来说，这是非常快的，但内核缓存有的时候可能满了，这就会导致你的程序挂起等待。因此，在一个生产系统中，我们应该避免对 stderr 写入过多的内容，因为它会堵塞真正需要的工作。

还需要注意的是，process.stderr 永远是 UTF-8 编码的数据流。不需要设置编码格式，你写入 process.stderr 的所有数据都会被当做 UTF-8 来处理。而且，你不能更改编码格式。

另外，Node 程序员要从操作系统读取的内容还包括了程序启动时的参数。argv 是包含命令行参数的数组，以 node 命令为第一个参数（例 5-22 和例 5-23）。

例 5-22　输出 argv 的简单脚本

```
console.log(process.argv);
```

例 5-23　运行例 5-22

```
Enki:~ $ node argv.js -t 3 -c "abc def" -erf       foo.js
[ 'node',
  '/Users/croucher/argv.js',
```

```
  '-t',
  '3',
  '-c',
  'abc def',
  '-erf',
  'foo.js' ]
Enki:~ $
```

这里需要注意几个问题。第一，process.argv 数组只是简单地把命令行内容以空格作分割得到的。如果两个参数之间包含多个空格，也只会被切分一次。检查空格的方法可以用正则表达式（regex）的写法 \s+，但这不包括引号内的空格，引号可以用来把多个词组合在一起。而且，还要注意第一个文件参数是如何被展开成全路径的。这意味着你可以传给命令行一个相对路径的文件名作为参数，它会在 argv 中显示成绝对路径。这对一些特殊字符也同样生效，比如用 ~ 来表示 home 目录，只有第一个参数会被这样展开。

argv 在编写命令行脚本的时候相当有用，但又有点太原始了。有几个社区项目对它进行了扩展，可以帮助你轻松编写命令行程序，包括功能自动启用、编写程序内帮助文档及其他高级功能。

4. 事件循环和计数器

如果你之前在浏览器里使用过 JavaScript 编程，就应该对 setTimeout() 很熟悉了。在 Node 里，我们有更加直接的方法来访问事件循环，并且可以推延工作，这些非常有用。process.nextTick() 创建了一个回调函数，它会在下一个 tick 或者事件循环下一次迭代时被调用。因为实现是使用队列的，所以它会取代其他事件。让我们在例 5-24 中进一步查看。

例 5-24　用 process.nextTick() 往事件循环队列里插入回调函数

```
> var http = require('http');
> var s = http.createServer(function(req, res) {
... res.writeHead(200, {});
... res.end('foo');
... console.log('http response');
... process.nextTick(function(){console.log('tick')});
... });
> s.listen(8000);
>
> http response
tick
http response
tick
```

这个例子创建了一个 HTTP 服务器。服务端监听请求事件的函数调用 process.

nextTick() 创建了一个回调函数。无论我们向 HTTP 服务器发起多少次请求，tick 每次都会出现在事件循环的下一个轮回中。nextTick() 回调函数不像其他回调函数那样是一个单独的事件，因此也不像一般回调函数那样异常脆弱，如例 5-25 和例 5-26 所示。

例 5-25　在其他代码异常之后，nextTick() 继续工作

```
process.on('uncaughtException', function(e) {
  console.log(e);
});

process.nextTick(function() {
  console.log('tick');
});
process.nextTick(function() {
  iAmAMistake();
  console.log('tock');
});
process.nextTick(function() {
  console.log('tick tock');
});
console.log('End of 1st loop');
```

例 5-26　例 5-25 运行结果

```
Enki:~ $ node process-next-tick.js
End of 1st loop
tick
{ stack: [Getter/Setter],
  arguments: [ 'iAmAMistake' ],
  type: 'not_defined',
  message: [Getter/Setter] }
tick tock
Enki:~ $
```

尽管故意制造了错误，但与其他在单个事件内的回调函数不同，tick 中的每一个函数都被隔离开了。让我们来看一下代码。首先，我们设置了异常监听器来捕获所有的异常。其次，调用 process.nextTick() 设置了几个回调函数。每一个回调函数都会输出到终端。但是，第二个函数有一个故意的错误。最后，我们在终端记录了一条消息。当 Node 运行这个程序的时候，它先处理了所有的代码，并且包括了输出 'End of 1st loop'。然后它按顺序调用了 nextTice() 中的回调函数。第一个 'tick' 输出后，我们抛出了异常，因为遇到了下一个 tick 中故意安放的错误。这个错误导致进程触发了一个 uncaughtException 事件，并使得我们的函数把错误输出到终端上。因为抛出了异常，'tock' 并没有在终端打印出来，但 'tick tock' 依然打印了，这是因为每次调用 nextTick() 的时候，回调函数都是在隔离中创建的。你可能会想到将要被触发的事件是在事件循环当前遍历的内部执行的。

而与其他事件相比，nextTick() 则是在事件循环的遍历开始前被调用的。最后，其他事件在事件循环内按顺序执行。

5.3.2 子进程

你可以使用 child_process 模块来为 Node 主进程创建子进程。因为 Node 的单进程只有一个事件循环，所以有时候创建子进程是很有用的。比如，你可能需要用此方法来更好地利用 CPU 的多核，而单个 Node 进程只能使用其中一个核。或者说，你可以用 child_process 来启动其他程序，然后与其交互。特别是当你在编写命令行脚本的时候，这会非常有用。

child_process 有两个主要的方法。spawn() 会创建一个子进程，并且有独立的 stdin、stdout 和 stderr 文件描述符。exec() 会创建子进程，并会在进程结束的时候以回调函数的方式返回结果。创建子进程的方法有很多种，其中一种依然是非阻塞的方式，而且不需要你写额外的代码来推动运行。

所有的子进程都有一些公共的属性。它们每个都包含了 stdin、stdout 和 stderr 的特性，正如我们在上一小节所讨论的那样。此外它们还有一个 pid 属性，它包含了该子进程的 OS 进程 ID。子进程在退出的时候会触发 exit 事件。其他 data 事件可以通过 child_process.stdin、child_process.stdout 和 child_process.stderr 的流方法获得。

1. child_process.exec()

让我们用最直观的使用情景来介绍 exec() 吧。使用 exec()，你可以创建一个子进程来运行其他程序（也可以是另外一个 Node 程序），然后在回调函数中返回执行的结果（例 5-27）。

例 5-27　用 exec() 调用 ls

```
var cp = require('child_process');

cp.exec('ls -l', function(e, stdout, stderr) {
  if(!e) {
    console.log(stdout);
    console.log(stderr);
  }
});
```

当调用 exec() 时，可以输入一个命令行指令让新创建的进程去执行。注意整个命令是一个字符串。如果你需要给命令传入参数，也需要将其包含在字符串里。在这个例子中，我们传给 ls 命令 -l 参数，用来指定输出格式为详细格式。你还可以使

用复杂的命令行功能，比如“|”来实现管道命令。Node 会返回管道中最后一个命令的结果。

回调函数接收 3 个参数：一个 error 对象、stdout 的结果和 stderr 的结果。注意调用的 `ls` 命令会运行在 Node 程序当前所在的工作目录中，你可以调用 `process.cwd()` 获得这个目录。

重要的是要了解第一个和第三个参数的区别。如果子进程返回了错误的状态码或者是有其他异常发生，error 对象就不会是 `null`。当子进程退出时，它会把状态传回给父进程。比如，在 Unix 中，0 是表示成功，大于 0 的 8 位数字则用来表示错误。error 对象也可以用来表示被调用的命令不满足 Node 对它的限制。当错误代码从子进程返回时，error 对象会包含错误代码和 stderr。但是，若一个子进程运行是成功的，stderr 中依然可以有数据。

`exec()` 的第二个参数可以是一个可选的配置对象。默认情况下，这个对象包含了如例 5-28 所示的属性。

例 5-28　`child_process.exec()` 的默认配置对象

```
var options = { encoding: 'utf8',
                timeout: 0,
                maxBuffer: 200 * 1024,
                killSignal: 'SIGTERM',
                setsid: false,
                cwd: null,
                env: null };
```

这些属性如下。

- `encoding`
  I/O 流输入字符的编码格式。

- `timeout`
  进程运行的时间，以毫秒为单位。

- `killSignal`
  当时间或 `Buffer` 大小超过限制时，用来终止进程的信号。

- `maxBuffer`
  stdout 或 stderr 允许最大的大小，以千字节为单位。

- `setsid`
  是否创建 Node 子进程的新会话。

- `cwd`

  为子进程初始化工作目录（`null` 表示使用当前的进程工作目录）。

- `env`

  进程的环境变量。所有的环境变量都可以从父进程继承。

让我们设置一些选项来给子进程一些限制吧。首先，我们限制响应数据的 `Buffer` 大小，如例 5-29 所示。

例 5-29　限制 `child_process.exec()` 调用的 `Buffer` 大小

```
> var child = cp.exec('ls', {maxBuffer:1}, function(e, stdout, stderr) {
... console.log(e);
... }
... );
> { stack: [Getter/Setter],
  arguments: undefined,
  type: undefined,
  message: 'maxBuffer exceeded.' }
```

在本例中，你可以看见我们设置了一个很小的 `maxBuffer`（只有 1kb），所以运行 `ls` 命令很快就耗尽所有的可用空间并且抛出错误。因此检查错误很重要，这让你能够用合理的方法来处理它们。因为你已经限制在 `child_process` 里，所以你不会希望由于访问了不存在的资源而导致真正的异常发生。如果 `child_process` 返回了一个错误，它的 `stdin` 和 `stdout` 属性就不可用了，因此如果再去访问它们将会抛出异常。

我们也可以在子进程运行超过一定时间后，把它终止掉，如例 5-30 所示。

例 5-30　`process.exec()` 调用时设置超时

```
> var child = cp.exec('for i in {1..100000};do echo $i;done',
... {timeout:500, killSignal:'SIGKILL'},
... function(e, stdout, stderr) {
...   console.log(e);
... });
> { stack: [Getter/Setter], arguments: undefined, type: undefined, message:
... }
```

这个例子定义了一个故意长时间运行的进程（在 shell 脚本中从 1 数到 100 000），但我们又设置了一个很短的超时。注意，我们还指定了 `killSignal`。默认的终止信号是 SIGTERM，但我们使用 SIGKILL 来展示这个功能。[3] 得到错误的返回时，注意一下其中的 `killed` 属性，它会告诉我们 Node 主动终止了该进程，并且它没有自行退出。这对于前一个例子也同样成立。因为它不是自己主动退出的，所以也没有 `code` 属性及其他关于系统错误的属性。

注 3：SIGKILL 可以在命令行用 kill -9 产生。

2. child_process.spawn()

spawn() 和 exec() 很像，但它是一个更加通用的方法。它要求你自己处理流和它们的回调函数。这让它的功能更加强大和灵活，但这也意味着需要编写更多的代码才能达到 exec() 那些一下子就能完成的功能。所以 spawn() 最常见的用途是用来在服务器开发中创建服务器程序的子模块，它也是人们使 Node 运行在一台机器的多个 CPU 核上的最常见方式。

虽然其功能和 exec() 一样，但 spawn() 的 API 还是有些差异的（例 5-31 和例 5-32）。第一个参数依然是让进程去开始运行的命令，但与 exec() 不同，它不再是一个命令字符串，而只是可执行程序。进程的参数以数组的形式作为第二个（可选的）参数传给 spawn()。这和 process.argv 的反向操作类似：不是把命令按空格分隔开，而是提供一个数组来以空格连接起来（join()）。

最后，spawn() 还可以接受一个选项数组作为最后一个参数。配置的部分属性与 exec() 的相同，我们马上会进一步介绍。

例 5-31　用 spawn() 启动子进程

```
var cp = require('child_process');

var cat = cp.spawn('cat');

cat.stdout.on('data', function(d) {
  console.log(d.toString());
});
cat.on('exit', function() {
  console.log('kthxbai');
});

cat.stdin.write('meow');
cat.stdin.end();
```

例 5-32　例 5-31 的运行结果

```
Enki:~ $ node cat.js
meow
kthxbai
Enki:~ $
```

在上面的例子中，我们使用了 Unix 程序的 cat 命令，它会把所有输入的内容都复制一遍并打印出来。你会看到，与 exec() 不同，我们没有直接给 spawn() 指定回调函数，因为期待使用子进程类提供的流事件来读取并发送数据。我们把子进程的实例命名为“cat”变量，然后就可以通过 cat.stdout 来设置子进程 stdout 流的事件监听器了。我们为 cat.stdout 设置了监听器来监控所有的 data 事件，并且对子进程本身设置了 exit 事件的监听器。通过其 child.stdin 流，就可以接着往子

进程的 stdin 中发送数据。这只是一个普通的可写数据流，但是，由于 cat 程序的行为特点，当我们关闭 stdin 的时候，子进程就会退出。这并非对所有程序都有效，但对于 cat 程序来说是有效的，因为它的存在只是为了把数据回显。

传给 spawn() 的配置内容并非和 exec() 完全一样，这是因为你需要对 spawn() 进行更多的手工操作。env、setsid 和 cwd 属性都是 spawn() 的可选项。还有 uid 和 gid，分别用来设置用户 ID 和组 ID。与 process 类似，设置 uid 或 gid 来修改用户名或用户组的名称会因为查找用户或用户组而短暂堵塞。spawn() 还比 exec() 多一个配置项，你可以设置自定义的文件描述符来传给新建立的子进程。让我们多花点时间在这个话题上吧，毕竟它有点复杂。

Unix 系统中的文件描述符是用来记录跟踪该程序正在对哪些文件进行操作的方法。因为 Unix 允许多个程序同时运行，所以需要有方法来确保这些程序在修改文件系统时不会不小心把别人的修改覆盖。文件描述符表是用来记录一个进程想要访问的所有文件信息的，内核可能会为了防止两个程序同时修改一个文件而把某个特定的文件锁住，当然还有其他管理功能。进程会从文件描述符表中查找某个文件对应的文件描述符，然后传给内核去访问该文件。文件描述符其实只是用一个整数来表示。

重要的一点是，文件描述符这个名字有点虚幻，因为它并不是单纯地表示文件。网络或其他 socket 一类的东西也是分配成文件描述符。Unix 的跨进程通信（IPC）socket 可以用来让进程间互相发消息，我们称它们为 stdin、stdout 和 stderr。当 spawn() 允许我们在创建新的子进程时指定文件描述符时，情况变得有趣起来。这意味着，不必由操作系统指派一个新的文件描述符，我们可以要求子进程与父进程一起共享一个已经存在的文件描述符。该文件描述符可以是一个连接在互联网的网络 socket，或者只是父进程的 stdin。但重点是，我们有了一个功能强大的方法来把工作分配给子进程了。

这是如何做到的呢？当传递 options 对象给 spawn() 时，我们可以指定 customFds 来把自己拥有的 3 个文件描述符传递给子进程，这样进程就不需要创立新的 stdin、stdout 和 stderr 文件描述符了（例 5-33 和例 5-34）。

例 5-33　把 stdin、stdout 和 stderr 传给子进程

```
var cp = require('child_process');

var child = cp.spawn('cat', [], {customFds:[0, 1, 2]});
```

例 5-34　运行例 5-33，并通过 stdin 用管道传入数据

```
Enki:~ $ echo "foo"
foo
```

```
Enki:~ $ echo "foo" | node

readline.js:80
    tty.setRawMode(true);
        ^
Error: ENOTTY, Inappropriate ioctl for device
    at new Interface (readline.js:80:9)
    at Object.createInterface (readline.js:38:10)
    at new REPLServer (repl.js:102:16)
    at Object.start (repl.js:218:10)
    at Function.runRepl (node.js:365:26)
    at startup (node.js:61:13)
    at node.js:443:3
Enki:~ $ echo "foo" | cat
foo
Enki:~ $ echo "foo" | node fds.js
foo
Enki:~ $
```

文件描述符 `0`、`1`、`2` 分别代表了 stdin、stdout 和 stderr。在例子中，我们创建了一个子进程，并从父进程给它传递 stdin、stdout 和 stderr。可以在命令行里进行连接测试。`echo` 命令可以打印出字符串 foo。如果直接把它用管道传给 node 程序（stdout 到 stdin），结果是出错。但是，我们可以把它传递给 `cat` 命令，它会把内容回显出来。同样，如果把内容通过管道传给运行脚本的 Node 程序，它也会把内容重复出来。这是因为我们将 Node 进程的 stdin、stdout 和 stderr 都与子进程中的 `cat` 程序绑定在一起了。当 Node 主进程从 stdin 得到数据的时候，它会传给 `cat` 子进程，并由 `cat` 程序把内容回传给共享的 stdout。要注意的一点是，一旦你把 Node 程序以这种方式连接起来，子进程就丢失了它的 `child.stdin`、`child.stdout` 和 `child.stderr` 的文件描述符引用。这是因为一旦把文件描述符传递给子进程，它们就会被复制，并且由内核来处理数据传递。因此，Node 并不是在进程与文件描述符之间（FD），所以你无法对这些数据流添加事件监听器（见例 5-35、例 5-36）。

例 5-35　当传递自定义文件描述符后，尝试访问这些文件描述符流失败

```
var cp = require('child_process');
var child = cp.spawn('cat', [], {customFds:[0, 1, 2]});
child.stdout.on('data', function(d) {
  console.log('data out');
});
```

例 5-36　测试结果

```
Enki:~ $ echo "foo" | node fds.js

node.js:134
        throw e; // process.nextTick error, or 'error' event on first tick
 foo
      ^
```

```
TypeError: Cannot call method 'on' of null
    at Object.<anonymous> (/Users/croucher/fds.js:3:14)
    at Module._compile (module.js:404:26)
    at Object..js (module.js:410:10)
    at Module.load (module.js:336:31)
    at Function._load (module.js:297:12)
    at Array.<anonymous> (module.js:423:10)
    at EventEmitter._tickCallback (node.js:126:26)
Enki:~ $
```

当指定了自定义的文件描述符，这些流就被显式地设置为 `null`，并且完全不能从父进程访问了。但在许多情况下这是有价值的，因为比起用 Node 的 `stream.pipe()` 把数据流连接起来，通过内核来分发要快很多。而且，stdin、stdout 和 stderr 并非仅有的几个值得用来连接子进程的文件描述符。一个常见的使用情境是把网络 socket 和一组子进程相连接，来利用多核的性能。

假设我们在创建一个网站或游戏服务器，或者任何需要处理大量流量的应用。我们有着强大的服务器，上面有一堆处理器，每个又有 2 个或 4 个核。假如只是简单地启动 Node 进程来运行代码，就只能用上一个核。虽然 CPU 通常不是 Node 程序的核心因素，但我们还是想尽量接近 CPU 的极限。此时我们可以把 Node 程序启动到不同的端口上，然后利用 Nginx 或 Apache 来进行负载均衡。但是，这样做并不优雅，而且要使用更多的软件。我们也可以让 Node 进程启动许多子进程，然后把请求分发给它们。这离理想解决方案已经很接近了，但是这个方法会出现一个单点故障，因为只有一个 Node 进程来分发所有的数据，这还不够理想。现在就是传递 custom FD 大显身手之时了。用传递主进程 stdin、stdout 和 stderr 同样的方法，我们可以创建其他 socket 并且把它们传给子进程。但因为我们传递的是文件描述符而不是消息，所以内核会负责处理分发。这意味着，即使依然需要有一个主 Node 进程，但是它不再需要承载所有的流量负荷了。

5.4 用assert来测试

`assert` 是为测试代码提供基础功能的核心库。Node 的断言功能与其他开发语言及环境所提供的功能很类似：允许你为对象或函数调用提出要求，并且在破坏断言的时候发出信息。这些方法都很容易使用，并能为代码单元测试提供许多便利。Node 自己的测试也是用 `assert` 编写的。

`assert` 的许多方法都是成对出现的，一个方法提供了正面的测试，另一个就提供反面的功能。比如例 5-37 演示的 `equal()` 和 `notEqual()`。这些方法接受两个参数，第一个是期待的值，第二个是实际的值。

例 5-37　assert 的基本功能

```
> var assert = require('assert');
> assert.equal(1, true, 'Truthy');
> assert.notEqual(1, true, 'Truthy');
AssertionError: Truthy
    at [object Context]:1:8
    at Interface.<anonymous> (repl.js:171:22)
    at Interface.emit (events.js:64:17)
    at Interface._onLine (readline.js:153:10)
    at Interface._line (readline.js:408:8)
    at Interface._ttyWrite (readline.js:585:14)
    at ReadStream.<anonymous> (readline.js:73:12)
    at ReadStream.emit (events.js:81:20)
    at ReadStream._emitKey (tty_posix.js:307:10)
    at ReadStream.onData (tty_posix.js:70:12)
>
```

这里最明显的就是 `assert` 方法不通过时，会抛出异常。这是测试套件的基本原则。当一个测试套件运行时，它应该只是运行，不会抛出异常。在这种情况下，测试会被认为是成功的。

只有几个断言函数，如 `equal()` 和 `notEqual()`，会检查相等（`==`）和不相等（`!=`）操作。这意味着其他的测试只会弱化地检查真值和假值（truthy 和 falsy，这是 Crockford 给它们起的名称）。简单而言，当测试作为一个布尔值时，假值包含了 false、0、空字符串（如 `""`）、`null`、`undefined` 和 `NaN`，所有其他值都为真值。一个像 `"false"` 这样的字符串是真值，一个包含 `"0"` 的字符串也是真值。而 `equal()` 和 `notEqual()` 可以用来比较两个简单对象的值（如字符串、数字）。但你需要仔细检查布尔值，以确保得到想要的结果。

`strictEqual()` 和 `notStrictEqual()` 方法检测两个数值是否相等时会采用 `===` 和 `!==`，这样可以确保测试时的 `true` 和 `false` 可分别被作为真和假来对待。例 5-38 中的 `ok()` 方法是用来测试一个对象是否为真值的简便方法，它会使用 `==` 来对比测试对象和 `true` 是否一样。

例 5-38　用 assert.ok() 测试某个对象是否为真值

```
> assert.ok('This is a string', 'Strings that are not empty are truthy');
> assert.ok(0, 'Zero is not truthy');
AssertionError: Zero is not truthy
    at [object Context]:1:8
    at Interface.<anonymous> (repl.js:171:22)
    at Interface.emit (events.js:64:17)
    at Interface._onLine (readline.js:153:10)
    at Interface._line (readline.js:408:8)
    at Interface._ttyWrite (readline.js:585:14)
    at ReadStream.<anonymous> (readline.js:73:12)
    at ReadStream.emit (events.js:81:20)
```

```
    at ReadStream._emitKey (tty_posix.js:307:10)
    at ReadStream.onData (tty_posix.js:70:12)
>
```

但通常你想要比较的内容并不是简单值，而是对象。JavaScript 并没有提供某种方法来让对象为自己定义相等运算符。即使它允许这样做，人们通常也不会定义运算符。所以 `deepEqual()` 和 `notDeepEqual()` 方法提供了深入比较两个对象值的方法。这些方法会进行若干测试，而无需太多的细节。如果任何一个检查失败了，测试就会抛出异常。首先检查的是若用简单的 `===` 操作来比较，两个值的结果是否相等。接着，检查一下它们的类型是否为 `Buffer`，如果是，则检查它们的长度，然后按字节对比。如果对象的类型按 `==` 运算符不匹配，它们就不可能相等。最后，如果比较的参数是对象类型，会进行更加严格的测试，如比较两个对象的原型、属性数量，然后对每个属性执行 `deepEqual()` 以进行递归比较。

这里需要重点指出，`deepEqual()` 和 `notDeepEqual()` 是非常有用的，但是代价可能很大。你应该只在需要的时候才使用它们。虽然这些方法都尝试先做最快速的测试，但可能需要花费较长的时间才能找到不一致的地方。如果你提供的对象更加精确，如用对象的某个属性来代替整个对象，就可以显著提高测试的性能。

接下来要介绍的 `assert` 方法是 `throws()` 和 `doesNotThrow()`。这些方法会检查指定的代码块是否会抛出异常。你可以检测指定的异常，或者是任意的异常是否抛出。这些方法都很直观，但有几个选项需要研究一下。

大家很容易忽略这些测试，但处理异常是编写健壮 JavaScript 代码的重要组成部分，所以你该使用这些测试来确保写出的代码在正确的地方抛出异常。第 3 章提供了更多关于处理异常的信息。

要把代码块传给 `throws()` 和 `doesNotThrow()`，需要把它们包含在一个没有参数的函数里（例 5-39）。待测试的异常是可选的，如果没有传入，`throws()` 会检查是否有异常发生，而 `doesNotThrow()` 会确保不抛出异常。如果指定了错误类型，`throws()` 会检查该指定的异常，并且只会抛出该类型的异常。如果任意其他的异常抛出来，或者指定的异常没有抛出，测试都不会通过。对于 `doesNotThrow()`，当指定了一个错误，如果有指定异常之外的任何异常抛出时，它都会继续运行。而如果指定的异常出现了，测试就会中断。

例 5-39　用 `assert.throws()` 和 `assert.doesNotThrow()` 检查异常处理

```
> assert.throws(
... function() {
...   throw new Error("Seven Fingers. Ten is too mainstream.");
... });
```

```
> assert.doesNotThrow(
... function() {
...   throw new Error("I lived in the ocean way before Nemo");
... });
AssertionError:   "Got unwanted exception (Error).."
    at Object._throws (assert.js:281:5)
    at Object.doesNotThrow (assert.js:299:11)
    at [object Context]:1:8
    at Interface.<anonymous> (repl.js:171:22)
    at Interface.emit (events.js:64:17)
    at Interface._onLine (readline.js:153:10)
    at Interface._line (readline.js:408:8)
    at Interface._ttyWrite (readline.js:585:14)
    at ReadStream.<anonymous> (readline.js:73:12)
    at ReadStream.emit (events.js:81:20)
>
```

有 4 种方法可用来指定要查看或避免的错误类型，我们可以传入以下类型。

- *比较函数*

  该函数只接收一个参数，即异常错误对象。在函数里比较传入的异常是否与你想查找的类型匹配，如果匹配则返回真，否则返回假。

- *正则表达式*

  函数库会根据正则表达式来比较错误消息是否匹配你的要求，采用的是 JavaScript 的 `regex.test()` 方法。

- *字符串*

  函数库会直接比较错误消息与指定的字符串。

- *对象构造类型*

  函数库会用 `typeof` 来对异常进行操作并测试。如果该测试在调用 `typeof` 时抛出错误，则认为异常类型匹配，这可以用来使 `throws()` 和 `doesNotThrow()` 变得更为灵活。

5.5 虚拟机

虚拟机（vm）模块让你可以运行任意一块代码，并得到运行结果。它提供了一些功能，可以修改指定代码运行的上下文。这很有用，比如可以用来作为人造沙箱。但是代码还是运行在同一个 Node 进程里，所以你依然需要小心行事。`vm` 和 `eval()` 类似，但提供了更多功能和更好的 API 来管理代码。然而，它不像 `eval()` 那样能提供与本地作用域互动的能力。

用 `vm` 运行代码有两种方法。第一种与使用 `eval()` 的方法类似，把代码内嵌运行；第二种是先把代码预编译成 `vm.Script` 对象。我们看一下例 5-40，它演示了如何

用 vm 内嵌运行代码。

例 5-40　用 vm 来运行代码

```
> var vm = require('vm');
> vm.runInThisContext("1+1");
2
```

到目前为止，vm 看起来都很像 eval()。我们给它传入一段代码，它就返回了结果。但是，vm 并不会像 eval() 那样改变本地作用域的内容。用 eval() 执行的代码会像真的嵌入在当前位置运行一样，并且替换掉 eval() 函数调用，而调用 vm 方法就不能这样作用于本地作用域了。所以说，eval() 会修改周围的上下文，但 vm 不会（例 5-41）。

例 5-41　使用 vm 和 eval() 在访问本地作用域时的区别

```
> var vm = require('vm'),
... e = 0,
... v = 0;
> eval(e=e+1);
1
> e
1
> vm.runInThisContext('v=v+1');
ReferenceError: v is not defined
    at evalmachine.<anonymous>:1:1
    at [object Context]:1:4
    at Interface.<anonymous> (repl.js:171:22)
    at Interface.emit (events.js:64:17)
    at Interface._onLine (readline.js:153:10)
    at Interface._line (readline.js:408:8)
    at Interface._ttyWrite (readline.js:585:14)
    at ReadStream.<anonymous> (readline.js:73:12)
    at ReadStream.emit (events.js:81:20)
    at ReadStream._emitKey (tty_posix.js:307:10)
>
> vm.runInThisContext('v=0');
0
> vm.runInThisContext('v=v+1');
1
>
0
```

我们创建了 e 和 v 两个变量。在 eval() 中使用变量 e 时，代码执行结束后，该结果会影响正文内容。但是当对变量 v 用 vm.runInThisContext() 尝试做同样的操作时，得到的结果是出现异常，因为对变量 v 的引用是在等号右边，而该变量还没有定义。eval() 是运行在当前作用域的，而 vm 不是。

vm 实际上会在每一个实例的内部，维护一套独立的本地上下文，并且能够保持状态。因此，当我们在 vm 的作用域内创建了变量 v，该变量就能够在同一个 vm 的后

续操作中有效，并且保持上一次调用时的状态。但是 vm 内的变量 v 并不会影响运行在主事件循环中的本地作用域。

此外，也可以传给 vm 一个已经存在的上下文内容。该上下文会作为默认的上下文使用。

例 5-42 使用了 vm.runInNewContext()，并以第二个参数作为上下文对象。该对象的作用域就成了我们用 vm 运行代码的上下文。如果我们继续把它传给不同的调用，此上下文就会被修改。而且，这个上下文能够被全局作用域使用。

例 5-42　传给 vm 上下文

```
> var vm = require('vm');
> var context = { alphabet:"" };
> vm.runInNewContext("alphabet+='a'", context);
'a'
> vm.runInNewContext("alphabet+='b'", context);
'ab'
> context
{ alphabet: 'ab' }
>
```

你也可以把代码编译成 vm.Script 对象（例 5-43）。这样就可以重复运行同一段代码，从而节省一些代码量。在运行的时候，你可以选择用哪个上下文来执行，这样就可以很方便地对不同的上下文执行同一段代码了。

例 5-43　用 vm 把代码编译成脚本对象

```
> var vm = require('vm');
> var fs = require('fs');
>
> var code = fs.readFileSync('example.js');
> code.toString();
'console.log(output);\n'
>
> var script = vm.createScript(code);
> script.runInNewContext({output:"Kick Ass"});
ReferenceError: console is not defined
    at undefined:1:1
    at [object Context]:1:8
    at Interface.<anonymous> (repl.js:171:22)
    at Interface.emit (events.js:64:17)
    at Interface._onLine (readline.js:153:10)
    at Interface._line (readline.js:408:8)
    at Interface._ttyWrite (readline.js:585:14)
    at ReadStream.<anonymous> (readline.js:73:12)
    at ReadStream.emit (events.js:81:20)
    at ReadStream._emitKey (tty_posix.js:307:10)
> script.runInNewContext({"console":console,"output":"Kick Ass"});
Kick Ass
```

这个例子从一个 JavaScript 文件中读取代码，里面包含一句简单的命令 console.log(output);。我们先把它编译成一个 script 对象，这样就能对此脚本执行 script.runInNewContext()，并且传入一个上下文。为了演示，我们故意触发一个错误。当运行 vm.runInNewContext() 时，你需要传入所引用的对象（如 console 对象），否则，即使是最基础的全局函数也不能使用。还需要注意的是，抛出异常的位置是 undefined:1:1。

所有的 vm 运行命令都可以把文件名作为可选的最后一个参数。它不会改变其功能，但是允许你设置出现错误时在消息里想要显示的文件名字。如果你从磁盘加载并运行了许多文件，这个功能就很有帮助，因为它能够告诉你哪个代码出现了错误。该参数是完全随意的，因此你可以采用任何有助于调试的字符串作为参数。

第6章 数据访问

与其他 Web 服务器类似，Node 需要通过存储来进行数据持久化。离开了持久化，你的网站就只是一个宣传性质的静态站点，也就没有必要使用 Node 了。在本章中，我们会讲到如何连接常见的开源数据库，以及如何保存和读取数据。

6.1 NoSQL和文档存储

下面介绍的这几个 NoSQL 数据库和文档存储系统在 Web 应用开发领域越来越受欢迎，而且与 Node 结合起来使用非常地简单。

6.1.1 CouchDB

CouchDB 提供了 JavaScript 环境下基于 MVCC[1] 的文档存储。在 CouchDB 里面添加或修改文档（记录）时，整个数据集都会保存到存储上，并且把老的版本标记为过时的。该记录的老版本内容依然会被整合到最新的版本里面去。每当创建了一个完整的新版本时，都会写入到连续的内存中，以便于更快地读取。因此 CouchDB 被称为“最终一致性”。在大型的、可扩展部署中，多个实例有时会把较老（未同步）的记录发送给客户端，但对记录的所有修改最终会合并到主实例中。

1. 安装

并非某个特定版本的 CouchDB 库才能访问数据库，但有些库提供了更高级的抽象，

注 1：MVCC 是多版本并发控制（multi-version concurrency control）的缩写。

并能使代码操作起来更为简便。我们还需要安装一个 CouchDB 的服务器来测试示例代码，这其实很容易完成。

(1) 安装 CouchDB。最新版本的 CouchDB 可以从 Apache 项目主页下载 http://couchdb.apache.org/download.html，不同平台的安装指导也能在其 wiki 页面 http://wiki.apache.org/couchdb/installation 上找到。

如果你使用的是 Windows，除了按提示的方法从源代码编译外，还可以找到二进制安装包。与其他许多 NoSQL 产品一样，在 Linux 这类系统上安装是最容易的，而且支持得更好，但你也可以另外选择自己熟悉的系统。

(2) 安装 CouchDB 的 Node 模块。使用 CouchDB 并不是必须用额外的模块，因为 CouchDB 通过 REST 方式把它所有的服务都开放了，后面会详细介绍。

2. 通过 HTTP 使用 CouchDB

CouchDB 的一大优点是，它的 API 真的全都是 HTTP 接口。因为 Node 能够很好地使用 HTTP，所以它使用 CouchDB 也非常容易。正因为这个原因，我们可以直接对数据库操作，而不需要借助于其他客户端。

例 6-1 演示了如何在当前安装的 CouchDB 中生成一个数据库列表。在这个例子里，并没有对 CouchDB 服务器进行身份验证或管理权限的限制。这对连接在互联网上的数据库来说绝不是个好主意，但如果单纯用于演示的目的，还是可以的。

例 6-1　通过 HTTP 获取 CouchDB 的数据库列表

```
var http = require('http');

http.createServer(function (req, res) {
  var client = http.createClient(5984, "127.0.0.1");
  var request = client.request("GET", "/_all_dbs");
  request.end();

  request.on("response", function(response) {
    var responseBody = "";

    response.on("data", function(chunk) {
      responseBody += chunk;
    });

    response.on("end", function() {
      res.writeHead(200, {'Content-Type': 'text/plain'});
      res.write(responseBody);
      res.end();
    });
  });
}).listen(8080);
```

客户端的连接是用 `http` 库创建的，这与其他的 http 连接并没有区别。因为 CouchDB 是 RESTful 的接口，所以不需要额外的传输协议。需要注意的是，`request.end()` 这行代码是在 `createServer` 方法内部执行的。如果缺少了这行代码，请求会被挂起。

如前所述，所有的 CouchDB 方法都是 HTTP 调用的。因此，创建和删除数据库分别是通过向服务器提交相应的 PUT 和 DELETE 语句来实现的，见例 6-2。

例 6-2　创建 CouchDB 数据库

```
var client = http.createClient(5984, "127.0.0.1")
var request = client.request("PUT", "/dbname");
request.end();

request.on("response", function(response) {
  response.on("end", function() {
    if ( response.statusCode == 201 ) {
      console.log("Database successfully created.");
    } else {
      console.log("Could not create database.");
    }
  });
});
```

在这里 `/dbname` 指代的是需要访问的资源。配合一个 PUT 命令，就是告诉 CouchDB 要创建一个新的数据库，名字叫 `dbname`。返回的 HTTP 代码 201 表示数据库创建成功。

如例 6-3 所示，删除资源用的是 PUT 的反操作：DELETE 命令。HTTP 返回代码 200 确认了该请求已成功执行。

例 6-3　删除 CouchDB 数据库

```
var client = http.createClient(5984, "127.0.0.1")
var request = client.request("DELETE", "/dbname");
request.end();

request.on("response", function(response) {
  response.on("end", function() {
    if ( response.statusCode == 200 ) {
      console.log("Deleted database.");
    } else {
      console.log("Could not delete database.");
    }
  });
});
```

这些元素单独使用并没有多大意义，但把它们组合起来，就能够作为一个基本的数据库管理工具了（虽然不是特别好用），它使用的方法如例 6-4 中所示。

例 6-4　一个简单的创建 CouchDB 数据库的方式

```
var http = require('http');
var qs = require('querystring');
var url = require('url');

var dbHost = "127.0.0.1";
var dbPort = 5984;

deleteDb = function(res, dbpath) {
  var client = http.createClient(dbPort, dbHost)
  var request = client.request("DELETE", dbpath);
  request.end();

  request.on("response", function(response) {
    response.on("end", function() {
      if ( response.statusCode == 200 ) {
        showDbs(res, "Deleted database.");
      } else {
        showDbs(res, "Could not delete database.");
      }
    });
  });
}

createDb = function(res, dbname) {
  var client = http.createClient(dbPort, dbHost)
  var request = client.request("PUT", "/" + dbname);
  request.end();

  request.on("response", function(response) {
    response.on("end", function() {
      if ( response.statusCode == 201 ) {
        showDbs(res, dbname + " created.");
      } else {
        showDbs(res, "Could not create " + dbname);
      }
    });
  });
}

showDbs = function(res, message) {
  var client = http.createClient(dbPort, dbHost);
  var request = client.request("GET", "/_all_dbs");
  request.end();

  request.on("response", function(response) {
    var responseBody = "";

    response.on("data", function(chunk) {
      responseBody += chunk;
    });

    response.on("end", function() {
      res.writeHead(200, {'Content-Type': 'text/html'});
```

```
        res.write("<form method='post'>");
        res.write("New Database Name: <input type='text' name='dbname' />");
        res.write("<input type='submit' />");
        res.write("</form>");
        if ( null != message ) res.write("<h1>" + message + "</h1>");

        res.write("<h1>Active databases:</h1>");
        res.write("<ul>");
        var dblist = JSON.parse(responseBody);
        for ( i = 0; i < dblist.length; i++ ) {
          var dbname = dblist[i];
          res.write("<li><a href='/" + dbname + "'>"+dbname+"</a></li>");
        }
        res.write("</ul>");
        res.end();
      });
    });
  };

  http.createServer(function (req, res) {
    if ( req.method == 'POST' ) {
      // 解析请求
      var body = '';
      req.on('data', function (data) {
        body += data;
      });
      req.on('end', function () {
        var POST = qs.parse(body);
        var dbname = POST['dbname'];
        if ( null != dbname ) {
          // 创建数据库
          createDb(res,dbname);
        } else {
          showDbs(res, "Bad DB name, cannot create database.");
        }
      });
    } else {
      var path = url.parse(req.url).pathname;
      if ( path != "/" ) {
        deleteDb(res,path);
      } else {
        showDbs(res);
      }
    }
  }).listen(8080);
```

3. 使用 node-couchdb

知道如何用 HTTP 来使用 CouchDB 是很有用的，但这个方法太过繁琐。虽然说不需要额外库是它的优势，但即便数据库的原生驱动实现起来非常简单，大多数程序员依然倾向于使用更高抽象级的工具。在本小节，我们会看一下 node-couchdb 包，它会简化 Node 与 CouchDB 间的接口。

使用 npm 来安装 CouchDB 的驱动：

```
npm install felix-couchdb
```

(1) 使用数据库。使用这个模块的第一个显著好处就是代码更简洁，如例 6-5 所示。

例 6-5　在 CouchDB 中创建一个表

```
var dbHost = "127.0.0.1";
var dbPort = 5984;
var dbName = 'users';

var couchdb = require('felix-couchdb');
var client = couchdb.createClient(dbPort, dbHost);
var db = client.db(dbName);

db.exists(function(err, exists) {
  if (!exists) {
    db.create();
    console.log('Database ' + dbName + ' created.');
  } else {
    console.log('Database ' + dbName + ' exists.');
  }
});
```

这个例子检查了是否有个数据库叫 users，如果不存在则创建一个新的。注意这里调用的 createClient 函数与之前使用 http 模块演示的函数有些相似。这不是偶然的，因为即使使用了更便捷的 CouchDB 模块，到最后你还是要使用 HTTP 来传输数据。

(2) 创建文档。在例 6-6 中，我们会在之前例子创建的 CouchDB 数据库中保存一个文档。

例 6-6　在 CouchDB 中创建一个文档

```
var dbHost = "127.0.0.1";
var dbPort = 5984;
var dbName = 'users';

var couchdb = require('felix-couchdb');
var client = couchdb.createClient(dbPort, dbHost);

var user = {
  name: {
    first: 'John',
    last: 'Doe'
  }
}

var db = client.db(dbName);

db.saveDoc('jdoe', user, function(err, doc) {
```

```
    if( err) {
      console.log(JSON.stringify(err));
    } else {
      console.log('Saved user.');
    }
  });
```

这个例子创建了一个用户，名为 John Doe，并以用户名 jdoe 作为标识保存在数据库里。注意，user 是一个 JSON 对象，并且直接传给了客户端，不需要其他工作来解析这些信息。

运行了这个例子后，该用户信息可以在浏览器里通过 http://127.0.0.1:5984/users/jdoe 访问。

(3) *读取文档*。文档一旦保存在了 CouchDB 里，就可以以对象的方式读取，如例 6-7 所示。

例 6-7　从 CouchDB 取回一条记录

```
var dbHost = "127.0.0.1";
var dbPort = 5984;
var dbName = 'users';

var couchdb = require('felix-couchdb');
var client = couchdb.createClient(dbPort, dbHost);

var db = client.db(dbName);

db.getDoc('jdoe', function(err,doc) {
  console.log(doc);
});
```

查询的输出内容是：

```
{ _id: 'jdoe',
  _rev: '3-67a7414d073c9ebce3d4af0a0e49691d',
  name: { first: 'John', last: 'Doe' }
}
```

这里分为如下 3 步执行。

(a) 使用 `createClient` 连接到数据库。

(b) 用客户端的 `db` 命令选择文档存储库。

(c) 用数据库的 `getDoc` 命令获取文档。

在这个例子里，ID 为 `jdoe` 的记录（之前例子中创建的）从数据库中取回。如果记录不存在（因为它被删除了或者没有插入），回调函数的 `error` 参数会包含错误信息。

(4) *更新文档*。更新文档使用的是与创建文档一样的 `saveDoc` 命令。如果 CouchDB 检测到有同样 ID 的记录存在，它会把旧的记录覆盖。

例 6-8 演示了如何从数据存储中读取一个文档然后更新它。

例 6-8　在 CouchDB 中更新一条记录

```
var dbHost = "127.0.0.1";
var dbPort = 5984;
var dbName = 'users';

var couchdb = require('felix-couchdb');
var client = couchdb.createClient(dbPort, dbHost);

var db = client.db(dbName);

db.getDoc('jdoe', function(err,doc) {
  doc.name.first = 'Johnny';
  doc.email = 'jdoe@johndoe.com';

  db.saveDoc('jdoe', doc );

  db.getDoc('jdoe', function(err,revisedUser) {
    console.log(revisedUser);
  });
});
```

本操作的输出为：

```
{ _id: 'jdoe',
  _rev: '7-1fb9a3bb6db27cbbbf1c74b2d601ccaa',
  name: { first: 'Johnny', last: 'Doe' },
  email: 'jdoe@johndoe.com'
}
```

这个例子从数据存储中读取关于 jdoe 用户的信息，增加了一个 email 地址和新的 first name，然后保存回 CouchDB。

注意在第一次读取数据之后的 `saveDoc` 和 `getDoc` 操作，`getDoc` 并不是放在 `saveDoc` 的回调函数里。因为 CouchDB 驱动会把命令保存在队列里，然后顺序执行，所以这个例子不会出现竞争状态，即在保存更新之前就先完成了文档读取操作。

(5) *删除文档*。要从 CouchDB 里删除文档，需要同时提供 ID 和版本号。好在从读取操作里很容易得到这些信息，见例 6-9。

例 6-9　在 CouchDB 中进行删除

```
var dbHost = "127.0.0.1";
```

```
var dbPort = 5984;
var dbName = 'users';

var couchdb = require('felix-couchdb');
var client = couchdb.createClient(dbPort, dbHost);

var db = client.db(dbName);

db.getDoc('jdoe', function(err,doc) {
  db.removeDoc(doc._id, doc._rev);
});
```

连接上 CouchDB 数据存储后，这里调用了 getDoc 命令，用来获得内部 ID（_id）和版本号（_rev）。一旦这些信息都拿到了，就可以调用 removeDoc 命令，它会发送一个 DELETE 请求给数据库。

6.1.2 Redis

Redis 是基于内存的 key-value 存储，并具备了持久化功能。如果你有使用 key-value 缓存（如 Memcache）的经验，这会让你觉得很熟悉。Redis 在性能和扩展性要求很高的情况下会被使用。在多数情况下，开发人员用它作为从关系型数据库（如 MySQL）中读取数据的缓存，但它能提供的功能其实很多。

除了 key-value 的存储能力，Redis 还提供了可供网络访问的共享内存、非阻塞的事件总线，还有订阅和发布的功能。

1. 安装

和许多其他的数据库类似，使用 Redis 需要先安装它的数据库应用程序和与它连接的 Node 驱动。

(1) 安装 Redis。Redis 是以源代码的方式提供的（http://redis.io/download）。配置起来并不需要做太多的工作，只要下载并按照网站的指示进行编译就可以了。

如果你使用的是 Windows，就要自己另外想办法了，因为目前 Redis 官方并不支持在 Windows 上运行。幸好，Redis 开发社区里一帮热心的支持者把 Redis 移植到了 Windows 上，比如使用 Cygwin 或者原生的编译版本。你可以在 https://github.com/dmajkic/redis 上找到用 MinGW 编译的原生 Windows 二进制文件。

(2) 安装 Redis 的 Node 模块。redis 模块可以从 GitHub 找到（https://github.com/mranney/node_redis），也可以用 npm 工具来安装：

```
npm install redis
```

或者，除了安装 Node 的 redis 模块之外，还可以安装更简约的 hiredis 库。

2. 基本功能

例 6-10 演示了通过 Node 对 Redis 进行基础的 set 和 get 操作。

例 6-10　Redis 里基础的 get 和 set 操作

```
var redis = require('redis'),
    client = redis.createClient();

client.on("error", function (err) {
    console.log("Error " + err);
});

console.log("Setting key1");
client.set("key1", "My string!", redis.print);
console.log("Getting key1");
client.get("key1", function (err, reply) {
    console.log("Results for key1:");
    console.log(reply);
    client.end();
});
```

这个例子先创建了一个 Redis 数据库的连接，并且设置了处理错误的回调函数。如果你没有运行 Redis 服务器，就会得到类似下面这样的错误：

```
Error Error: Redis connection to 127.0.0.1:6379 failed - ECONNREFUSED,
Connection refused
```

> 注意，这个例子里缺少了一些回调函数。如果需要在数据库写入操作之后马上读取，就应该使用更安全的方法——回调函数，这样才能确保代码按照正确的顺序执行。

在连接打开之后，客户端设置了一个基本数据，然后从数据库中把这些值读回来。库提供了与 Redis 基本命令一样的函数（`set`、`hset`、`get`）。Redis 把 `set` 命令传输过来的数据作为字符串使用，允许最大到 512 MB。

3. 哈希

哈希是包含多个 key 的对象，例 6-11 中每次设置了一个 key 的内容。

例 6-11　每次设置一个 hash 值

```
var redis = require('redis'),
    client = redis.createClient();

client.on("error", function (err) {
    console.log("Error " + err);
});

console.log("Setting user hash");
```

```
client.hset("user", "username", "johndoe");
client.hset("user", "firstname", "john");
client.hset("user", "lastname", "doe");

client.hkeys("user", function(err,replies) {
    console.log("Results for user:");
    console.log(replies.length + " replies:");
    replies.forEach(function (reply, i) {
        console.log(i + ": " + reply );
    });
    client.end();
});
```

例 6-12 演示了如何一次设置多个 key 的内容。

例 6-12　同时设置多个 hash 值

```
var redis = require('redis'),
    client = redis.createClient();

client.on("error", function (err) {
    console.log("Error " + err);
});

console.log("Setting user hash");
client.hmset("user", "username", "johndoe", "firstname", "john",
"lastname", "doe");

client.hkeys("user", function(err,replies) {
    console.log("Results for user:");
    console.log(replies.length + " replies:");
    replies.forEach(function (reply, i) {
        console.log(i + ": " + reply );
    });
    client.end();
});
```

当然，我们可以采用一种对程序员更加友好的方式来完成同样的功能，使用一个对象，而不是把每个 key、value 拆成列表，如例 6-13 所示。

例 6-13　使用对象来设置多个 hash 值

```
var redis = require('redis'),
    client = redis.createClient();

client.on("error", function (err) {
    console.log("Error " + err);
});

var user = {
   username: 'johndoe',
   firstname: 'John',
   lastname: 'Doe',
```

```
        email: 'john@johndoe.com',
        website: 'http://www.johndoe.com'
    }

    console.log("Setting user hash");
    client.hmset("user", user);

    client.hkeys("user", function(err,replies) {
        console.log("Results for user:");
        console.log(replies.length + " replies:");
        replies.forEach(function (reply, i) {
            console.log(i + ": " + reply );
        });
        client.end();
    });
```

除了可以手动把每一项都提供给 Redis，你还可以把整个对象传入 `hmset`，它会把内容解析出来并将正确的信息发送到 Redis 服务器。

> 需要注意，添加多个内容对象时，使用的是 `hmset` 而不是 `hset`。这里有个常见的错误，就是会忘记一个对象其实包含了多个值。

4. 列表

可以将列表类型想象成一个 key 包含了多个值（见例 6-14）。因为可以往列表的头部和尾部都添加内容，所以这些集合很适合用来展示有序事件，比如记录那些用户最近获得了某项荣誉。

例 6-14　在 Redis 中使用列表

```
    var redis = require('redis'),
        client = redis.createClient();

    client.on("error", function (err) {
        console.log("Error " + err);
    });

    client.lpush("pendingusers", "user1" );
    client.lpush("pendingusers", "user2" );
    client.lpush("pendingusers", "user3" );
    client.lpush("pendingusers", "user4" );

    client.rpop("pendingusers", function(err,username) {
      if( !err ) {
        console.log("Processing " + username);
      }
      client.end();
    });
```

例子的输出是：

```
Processing user1
```

这个例子演示了用 Redis 的 `list` 命令实现一个先入先出队列（FIFO）。现实中注册系统会用到 FIFO：当涌入的注册请求数量超过了实时处理的能力时，注册信息会被转移到队列中去，以便在主程序外处理。注册请求会按照接收的顺序处理，但主程序不会因为需要处理真正的创建记录和介绍工作（如发送欢迎邮件）而变慢。

5. 集合

当需要一个没有重复内容的列表时，使用集合，如例 6-15 所示。

例 6-15　使用 Redis set 命令

```
var redis = require('redis'),
    client = redis.createClient();

client.on("error", function (err) {
    console.log("Error " + err);
});

client.sadd( "myteam", "Neil" );
client.sadd( "myteam", "Peter" );
client.sadd( "myteam", "Brian" );
client.sadd( "myteam", "Scott" );
client.sadd( "myteam", "Brian" );

client.smembers( "myteam", function(err, members) {
  console.log( members );
  client.end();
});
```

输出是：

```
[ 'Brian', 'Scott', 'Neil', 'Peter' ]
```

即使 Brian 被添加到列表两次，它也只会被插入一次。在实际中，完全可能出现两个成员都叫 Brian 的情况，这就提醒我们需要在使用的时候确保插入的值是唯一的，否则集合会导致意外的结果，比如你得到的数量少于预期，是因为把重复的项目移除了。

6. 有序集合

和普通集合一样，有序集合也不允许重复成员。有序集合增加了权重的概念，允许在数据上进行基于分数的操作，比如排行榜、最高分和内容表。

美国减肥真人秀节目《超级减肥王》（*The Biggest Loser*）的制作方真是有序集合的

粉丝。在系列第 11 季中，比赛选手根据他们的年龄被分为 3 组。在直播中，选手们需要通过查看彼此衬衫上的数字来进行原始的排序，在烈日下按升序站成一列。如果其中一个选手带上装备 Node 及 Redis 的笔记本到比赛现场，他就可以写个小程序轻松地得到答案了，如例 6-16 所示。

例 6-16　使用 Redis 来进行列表排序

```
var redis = require('redis'),
    client = redis.createClient();

client.on("error", function (err) {
    console.log("Error " + err);
});

client.zadd( "contestants", 60, "Deborah" );
client.zadd( "contestants", 65, "John" );
client.zadd( "contestants", 26, "Patrick" );
client.zadd( "contestants", 62, "Mike" );
client.zadd( "contestants", 24, "Courtney" );
client.zadd( "contestants", 39, "Jennifer" );
client.zadd( "contestants", 26, "Jessica" );
client.zadd( "contestants", 46, "Joe" );
client.zadd( "contestants", 63, "Bonnie" );
client.zadd( "contestants", 27, "Vinny" );
client.zadd( "contestants", 27, "Ramon" );
client.zadd( "contestants", 51, "Becky" );
client.zadd( "contestants", 41, "Sunny" );
client.zadd( "contestants", 47, "Antone" );
client.zadd( "contestants", 40, "John" );

client.zcard( "contestants", function( err, length ) {
  if( !err ) {
    var contestantCount = length;
    var membersPerTeam = Math.ceil( contestantCount / 3 );
    client.zrange( "contestants", membersPerTeam * 0, membersPerTeam * 1 - 1,
      function(err, values) {
        console.log('Young team: ' + values);
      });
    client.zrange( "contestants", membersPerTeam * 1, membersPerTeam * 2 - 1,
      function(err, values) {
        console.log('Middle team: ' + values);
      });
    client.zrange( "contestants", membersPerTeam * 2, contestantCount,
      function(err, values) {
        console.log('Elder team: ' + values);
        client.end();
      });
  }
});
```

结果是：

```
Young team: Courtney,Jessica,Patrick,Ramon,Vinny
```

```
Middle team: Jennifer,John,Sunny,Joe,Antone
Elder team: Becky,Deborah,Mike,Bonnie
```

往有序集合添加成员的方法和普通集合的操作方法一样，但需要增加一项权重，这样可以很容易地做分割，如例子中所示。知道每个组的成员都年纪相近之后，要从一个排序列表中得到 3 个队伍，只要从集合中直接抽取 3 个相等的组就行了。因为比赛选手的数量（14）并不能被 3 整除，因此最后一组只有 4 个成员。

7. 订阅

Redis 支持发布 – 订阅（pub-sub）消息模型，允许发送者（发布者）往频道里添加消息，然后由匿名的接收者（订阅者）使用（例 6-17）。订阅者注册他们感兴趣的领域（频道），Redis 就会把相关的消息推送给他们。发布者不需要注册到特定的频道，或者在发送的时候不需要有监听的订阅者。Redis 会做好中间商，这提供了很大的便利，因此发布者和订阅者无需知道对方的信息。

例 6-17　使用 Redis 订阅与发布

```
var redis = require("redis"),
    talkativeClient = redis.createClient(),
    pensiveClient = redis.createClient();

pensiveClient.on("subscribe", function (channel, count) {
  talkativeClient.publish( channel, "Welcome to " + channel );
  talkativeClient.publish( channel, "You subscribed to " + count + "
channels!" );
});

pensiveClient.on("unsubscribe", function(channel, count) {
  if (count === 0) {
    talkativeClient.end();
    pensiveClient.end();
  }
});

pensiveClient.on("message", function (channel, message) {
  console.log(channel + ': ' + message);
});

pensiveClient.on("ready", function() {
  pensiveClient.subscribe("quiet channel", "peaceful channel", "noisy
channel" );
  setTimeout(function() {
    pensiveClient.unsubscribe("quiet channel", "peaceful channel", "noisy
channel" );
  }, 1000);
});
```

结果是：

```
quiet channel: Welcome to quiet channel
quiet channel: You subscribed to 1 channels!
peaceful channel: Welcome to peaceful channel
peaceful channel: You subscribed to 2 channels!
noisy channel: Welcome to noisy channel
noisy channel: You subscribed to 3 channels!
```

这个例子列举了两个客户端的情况，一个是安静且考虑周到的，而另一个会把它周围的信息广播给任何一个听众。深沉的客户端订阅了 3 个频道，quiet、peaceful 和 noisy，健谈的客户端为每个订阅者都发送欢迎来到该频道的信息，并记录了当前活跃订阅数。

在订阅一秒之后，深沉的客户端取消了所有 3 个频道的订阅。当 `unsubscribe` 命令检查到没有任何活跃订阅的时候，两个客户端都断开了与 Redis 的连接，并且退出程序。

8. Redis 安全性

Redis 支持密码验证。要添加密码，需要编辑 Redis 的配置文件，增加一句 `requirepass`，如例 6-18 所示。

例 6-18　Redis 密码配置例子

```
################################ SECURITY #################################

# Require clients to issue AUTH <PASSWORD> before processing any other
# commands.  This might be useful in environments in which you do not
  trust
# others with access to the host running redis-server.
#
# This should stay commented out for backward compatibility and because
  most
# people do not need auth (e.g., they run their own servers).
#
requirepass hidengoseke
```

一旦 Redis 重启，它就只会对使用 hidengoseke 作为密码授权的客户端服务了（例 6-19）。

例 6-19　登录 Redis

```
var redis = require('redis'),
    client = redis.createClient();

client.auth("hidengoseke");
```

auth 命令必须在其他操作之前执行。客户端会保存密码，并且在重新连接的时候使用。

注意这里没有用户名或多个密码。Redis 并没有提供用户管理功能，因为这会增加开销。因此，期望系统管理员能够用其他方法保护好他们的服务器，比如把 Redis 的端口从外界隔离开，只让可靠的用户在内部访问。

一些“危险”的命令可以改名或者彻底移除。比如，你可能永远都不会使用 CONFIG 命令。在这种情况下，可以修改配置文件，要么把它改名为其他安全的名称，要么把它禁用以避免不希望的使用。这两个方法都在例 6-20 中展示出来。

例 6-20　将 Redis 命令改名

```
# Change CONFIG command to something obscure
rename-command CONFIG 923jfiosflkja98rufadskjgfwefu89awtsga09nbhsdalkjf3p49

# Clear CONFIG command, so no one can use it
rename-command CONFIG ""
```

6.1.3　MongoDB

因为 Mongo 提供了 JavaScript 环境下的 BSON 对象存储（一种 JSON 的二进制变种），因此从 Node 去读写数据都非常高效。Mongo 把传入的数据保存在内存里，因此很适合高并发写操作的情况。它的每个新版本都不断提高了集群、复制和分片的功能。

因为写入的数据保存在内存中，所以往 Mongo 插入数据是非阻塞的，因此也很适合用来记录操作和远程数据。Mongo 支持在查询里嵌入 JavaScript 函数，还能进行 MapReduce 查询，所以读取数据的时候功能也很强大。

使用 MongoDB 的文档型存储，你可以把子记录保存在母记录的内部。比如，一篇博客及其相关的所有评论都能保存在一条记录里，这样读取起来就非常快。

1. MongoDB 原生驱动

Christian Kvaleim 编写的原生 MongoDB 驱动（https://github.com/christkv/node-mongodb-native）提供了访问 MongoDB 的非阻塞方式。之前的版本提供了 C/C++ 的 BSON 解析器和序列化功能，但因为 JavaScript 版本的性能提高，所以这套老的模块被弃用了。

原生 MongoDB 驱动的好处是，你可以用来精确地控制 MongoDB 连接的操作。

(1) 安装。要安装此驱动，需要运行以下的命令：

```
npm install mongodb
```

mongodb 是为了避免与后面要使用的 mongo 混淆。

(2) 数据类型。Node 的 MongoDB 驱动支持的数据类型列在表 6-1 中。

表6-1：MongoDB支持的数据类型

类　　型	描　　述	例　　子
Array	数据列表	`scardsInHand: [9,4,3]`
Boolean	真 / 假条件	`hasBeenRead: false`
Code	一段能在数据库中执行的 JavaScript 代码	`new BSON.Code('function quotient( dividend, divisor ) { return divisor == 0 ? 0 : divident / divisor; }');`
Date	表示当前日期和时间	`lastUpdated: new Date()`
DBRef	数据库引用 *	`bestFriendId: new BSON.DBRef('users', friendObjectId)`
Integer	一个整数（非十进制）数	`pageViews: 50`
Long	一个长整型值	`starsInUniverse = new BSON.Long(10000000000000000000000000);`
Hash	一个键值字典	`userName: {'first': 'Sam', 'last': 'Smith'}`
Null	一个空值	`bestFriend: null`
Object ID	MongoDB 用来索引对象的 12byte 代码，表示 24 位 16 进制字符串	`myRecordId: new BSON.ObjectId()`
String	一个 JavaScript 字符串	`fullName: 'Sam Smith'`

* 因为 MongoDB 是非关系型数据库，所以它并不支持 `join` 操作。数据类型 `DBRef` 是客户端用来实现逻辑关系 `join` 的。

(3) 写入记录。正如前面所说，往 MongoDB 集合中写入记录需要在 Node 中创建一个 JSON 对象，然后直接往 Mongo 中打印进去。例 6-21 演示了如何创建一个用户对象，并将其保存到 MongoDB 里。

例 6-21　连接到 MongoDB 并写入一条记录

```
var mongo = require('mongodb');
var host = "localhost";
var port = mongo.Connection.DEFAULT_PORT;
var db = new mongo.Db('node-mongo-examples', new mongo.Server(host, port,
{}), {});

db.open(function(err,db) {
  db.collection('users', function(err,collection) {
    collection.insert({username:'Bilbo',firstname:'Shilbo'}, function(err,
docs) {
```

```
      console.log(docs);
      db.close();
    });
  });
});
```

输出是：

```
[ { username: 'Bilbo',
    firstname: 'Shilbo',
    _id: 4e9cd8204276d9f91a000001 } ]
```

2. Mongoose

Node 使用 Mongoose 库能够支持大量的 Mongo 操作。与原生的驱动相比，Mongoose 是一个表达更加清楚的环境，让模型和架构更加直观。

(1) 安装。最快捷地安装和运行 Mongoose 的方法是用 npm 来安装它：

```
npm install mongo
```

或者，你可以从源代码下载到最新的版本，然后按照 Mongoose 项目主页（http://mongoosejs.com）的方法来编译。

(2) 定义结构（schema）。使用 MongoDB 的时候，不需要像关系数据库那样定义数据的结构。每当需求变更或者需要保存新的信息时，只要把包含新信息的记录保存进去，就能马上查询使用了。你可以把旧数据转换成新的字段，包含默认值或为空值，但 MongoDB 并不要求这样做。

虽然结构对于 MongoDB 来说并不重要，但它有助于人们理解数据的内容，并包含了该领域数据的操作规则。Mongoose 的作用在于，它使用了人可读的结构描述，提供了简洁的数据库交互接口。

什么是结构？许多程序员会想成是定义数据结构模型的术语，但较少会想到这些模型代表的底层数据库。SQL 数据库中的一张表需要在写入数据前先创建好，表中的每一个字段都与你的模型相匹配。该结构（指数据库中模型的定义）是与你的程序分别创建的，因此结构先于数据存在。

人们常说 MongoDB（及其他 NoSQL 数据库）是无结构的，因为它不要求显式地定义存储数据的数据结构。实际上，MongoDB 是有结构的，但并不由它保存的数据所决定。你可以在程序运行了几个月后对模型增加新的属性，但不必重新为之前已经保存的数据定义结构，就能对新的字段进行查询。

例 6-22 演示了如何为一个文章数据库定义样例结构，以及每种模型的类型该如何保存信息。再次说明，Mongo 并不强制执行结构，但程序员需要在自己的程序里定义统一的访问模式。

例 6-22　用 Mongoose 定义结构

```
var mongoose = require('mongoose')

var Schema   = mongoose.Schema,
    ObjectId = Schema.ObjectId

var AuthorSchema = new Schema({
    name: {
        first   : String,
        last    : String,
        full    : String
    },
    contact: {
        email   : String,
        twitter : String,
        google  : String
    },
    photo       : String
});

var CommentSchema = new Schema({
    commenter   : String,
    body        : String,
    posted      : Date
});

var ArticleSchema = new Schema({
    author      : ObjectId,
    title       : String,
    contents    : String,
    published   : Date,
    comments    : [CommentSchema]
});

var Author = mongoose.model('Author', AuthorSchema);
var Article = mongoose.model('Article', ArticleSchema);
```

(3) 操作集合（collection）。Mongoose 允许直接操作对象数据集合，如例 6-23 所示。

例 6-23　用 Mongoose 读写记录

```
mongoose.connect('mongodb://localhost:27017/upandrunning', function(err){
  if (err) {
    console.log('Could not connect to mongo');
  }
});

newAuthor.save(function(err) {
```

```
    if (err) {
      console.log('Could not save author');
    } else {
      console.log('Author saved');
    }
  });

  Author.find(function(err,doc){
    console.log(doc);
  });
```

这个例子在数据库里保存了一条作者信息，然后把所有作者在屏幕上打印出来。

(4) 性能。使用 Mongoose 的时候，你不需要自己维护 MongoDB 的连接，因为所有的结构定义和查询都会被缓存起来，直到真的连接使用。这点很重要，这也是 Mongoose 提供 Node 服务的重要方法。通过一次性把所有“正在执行的”命令提交到 Mongo，你可以限制使用时间，减少需要处理的回调函数，从而大大提升程序能够进行的操作的数量。

6.2 关系型数据库

我们有许多的理由依旧使用传统的 SQL 数据库，而 Node 对流行的开源数据库都有模块支持。

6.2.1 MySQL

MySQL 成为开源世界的主力军是有原因的：它免费提供了与大型商用数据库一样的众多功能。当前，MySQL 拥有很高的性能和丰富的功能。

1. 使用 NodeDB

node-db 模块提供了常用数据库系统的原生代码接口，包括与 MySQL 的接口。它通过该模块公开的通用 API 来给 Node 使用。虽然 node-db 不止支持 MySQL 一家，但本节将集中讲述如何在应用代码中使用 MySQL。自从 Oracle 收购了 Sun 公司，MySQL 及其社区的未来走向已受到很多猜测。有些组织主张迁移到直接替代品，如 MariaDB，或者是彻底更换到其他的关系型数据库管理系统（RDBMS）。虽然 MySQL 不会很快消失，但你需要判断它是否是工作的最佳选择。

(1) 安装。MySQL 客户端开发库是 Node 数据库模块必选的依赖项。在 Ubuntu，你可以使用 apt 命令安装这些库：

```
sudo apt-get install libmysqlclient-dev
```

然后使用 npm 来安装名为 db-mysql 的包：

```
npm install -g db-mysql
```

要运行本部分的示例代码，需要运行一个名为 upandrunning 的数据库，并且提供一个账号，其用户名和密码都是 dev。以下的脚本会创建基本的数据库表和结构：

```
DROP DATABASE IF EXISTS upandrunning;

CREATE DATABASE upandrunning;

GRANT ALL PRIVILEGES ON upandrunning.* TO 'dev'@'%' IDENTIFIED BY 'dev';

USE upandrunning;

CREATE TABLE users(
   id int auto_increment primary key,
   user_login varchar(25),
   user_nicename varchar(75)
);
```

(2) 选择。例 6-24 演示了如何从 WordPress 用户表中选出所有的 ID 和 user_name 列的内容。

例 6-24　从 MySQL 选出数据

```
var mysql = require( 'db-mysql' );

var connectParams = {
  'hostname': 'localhost',
  'user': 'dev',
  'password': 'dev',
  'database': 'upandrunning'
}

var db = new mysql.Database( connectParams );

db.connect(function(error) {
  if ( error ) return console.log("Failed to connect");

  this.query()
    .select(['id', 'user_login'])
    .from('users')
    .execute(function(error, rows, columns) {
      if ( error ) {
        console.log("Error on query");
      } else {
        console.log(rows);
      }
    });
});
```

你也许能够猜到，这里执行的效果等价于 SQL 命令 `SELECT id, user_login FROM users`。输出为：

```
{ id: 1, user_login: 'mwilson' }
```

(3) 插入。插入数据和选择数据的方法类似，因为命令都是以同样的方式串联起来的。例 6-25 演示了如何进行 `INSERT INTO users ( user_login ) VALUES ( 'newbie');` 的操作。

例 6-25　插入到 MySQL 中

```
var mysql = require( 'db-mysql' );

var connectParams = {
  'hostname': 'localhost',
  'user': 'dev',
  'password': 'dev',
  'database': 'upandrunning'
}

var db = new mysql.Database( connectParams );

db.connect(function(error) {
  if ( error ) return console.log("Failed to connect");

  this.query()
    .insert('users', ['user_login'], ['newbie'])
    .execute(function(error, rows, columns) {
      if ( error ) {
        console.log("Error on query");
        console.log(error);
      }
      else console.log(rows);
    });
});
```

输出是：

```
{ id: 2, affected: 1, warning: 0 }
```

`.insert` 命令接受下列 3 个参数。

- 表的名称。
- 待插入列的名称。
- 插入每列的对应值。

数据库驱动会处理好转义，并把数据类型转换成列需要的值。所以你不需要在传给这个模块的代码上担心 SQL 注入攻击。

(4) 更新。与选择和插入一样，更新操作也是依赖链式函数来生成等价的 SQL 操作。例 6-26 展示的是使用查询条件来限定更新的目标，而不是对整个数据库表的所有记录都作精心修改。

例 6-26　在 MySQL 中更新数据

```
var mysql = require( 'db-mysql' );

var connectParams = {
  'hostname': 'localhost',
  'user': 'dev',
  'password': 'dev',
  'database': 'unandrunning'
}

var db = new mysql.Database( connectParams );

db.connect(function(error) {
  if ( error ) return console.log("Failed to connect");

  this.query()
         .update('users')
         .set({'user_nicename': 'New User' })
         .where('user_login = ?', [ 'newbie' ])
    .execute(function(error, rows, columns) {
      if ( error ) {
        console.log("Error on query");
        console.log(error);
      }
      else console.log(rows);
  });
});
```

输出是：

```
{ id: 0, affected: 1, warning: 0 }
```

更新一行数据包含下面 3 个部分。

- .update 命令，接受表名（本例为 users）作为参数。
- .set 命令，使用 key-value 对象来确定哪一列需要修改，以及它们的新值是多少。
- .where 命令，告诉 MySQK 该如何过滤需要修改的记录。

(5) 删除。如例 6-27 所示，删除与更新非常类似，唯一的不同是，在删除的时候，不需要指定哪一列有更新。如果没有指定 where 条件，那么整个表的数据都将被删除干净。

例 6-27　从 MySQL 中删除数据

```
var mysql = require( 'db-mysql' );
```

```
var connectParams = {
  'hostname': 'localhost',
  'user': 'dev',
  'password': 'dev',
  'database': 'upandrunning'
}

var db = new mysql.Database( connectParams );

db.connect(function(error) {
  if ( error ) return console.log("Failed to connect");

  this.query()
    .delete()
    .from('users')
    .where('user_login = ?', [ 'newbie' ])
    .execute(function(error, rows, columns) {
      if ( error ) {
        console.log("Error on query");
        console.log(error);
      }
      else console.log(rows);
    });
});
```

输出是：

```
{ id: 0, affected: 1, warning: 0 }
```

`.delete` 命令和 `.update` 命令类似，唯一的不同是它不接受任何列的名字和数据。在这个例子里，where 条件里使用了通配符：`'user_login = ?'`。代码中的问号会被 `user_login` 参数替换掉，然后再执行。第二个参数是个数组，因为如果使用了多个问号，数据库驱动就需要从这个参数中按顺序取值使用。

2. Sequelize

Sequelize 是一个对象关系映射（ORM），它把之前部分介绍的重复工作简化了。你可以使用 Sequelize 来定义数据库与程序间共享的对象，这样就不需要为每个操作写查询语句，而是直接通过操作这些对象来写入或读取数据库。当你在进行维护或者增加新数据列的时候，这绝对能节省不少时间，而且能在整个数据管理工作中减少错误。Sequelize 支持通过 `npm` 安装：

```
npm install sequelize
```

上一小节的例子中已经创建好了数据库和示例用户，现在是时候在数据库里创建 `Author` 实体了（例 6-28）。Sequelize 替你处理了创建操作，所以此时不需要手动执行任何 SQL 操作。

例 6-28　用 Sequelize 创建实例

```
var Sequelize = require('sequelize');

var db = new Sequelize('upandrunning', 'dev', 'dev', {
  host: 'localhost'
});

var Author = db.define('Author', {
  name: Sequelize.STRING,
  biography: Sequelize.TEXT
});

Author.sync().on('success', function() {
  console.log('Author table was created.');
}).on('failure', function(error) {
  console.log('Unable to create author table');
});
```

输出是：

```
Executing: CREATE TABLE IF NOT EXISTS`Authors` (`name` VARCHAR(255),
`biography`
TEXT,`id` INT NOT NULL auto_increment ,`createdAt` DATETIME NOT NULL,
`updatedAt`
DATETIME NOT NULL, PRIMARY KEY (`id` )) ENGINE=InnoDB;
Author table was created.
```

在本例中，`Author` 被定义为一个包含了 `name` 字段和 `biograph` 字段的实例。在输出中能够看见，Sequelize 添加了一个自增的主键列、`createdAt` 列和 `updatedAt` 列。这是许多 ORM 采用的典型方法，提供了标准化的接口让 Sequelize 能够关联和处理你的数据。

Sequelize 与本章之前介绍的库有所区别，它是基于监听事件驱动的架构，而不是其他地方采用的回调函数驱动的架构。这意味着，你需要在每个操作之后同时监听成功和失败事件，而不是在操作返回值中包含成功或失败的信息。

例 6-29 创建了多对多关系的两个表，操作的顺序如下。

(1) 设置实例的结构。

(2) 把结构（schema）与真实的数据库进行同步。

(3) 创建并保存一个 `Book` 对象。

(4) 创建并保存一个 `Author` 对象。

(5) 建立 `author` 和 `book` 之间的关系。

例 6-29　用 Sequelize 保存记录和关系

```
var Sequelize = require('sequelize');

var db = new Sequelize('upandrunning', 'dev', 'dev', {
  host: 'localhost'
});

var Author = db.define('Author', {
  name: Sequelize.STRING,
  biography: Sequelize.TEXT
});

var Book = db.define('Book', {
  name: Sequelize.STRING
});

Author.hasMany(Book);
Book.hasMany(Author);

db.sync().on('success', function() {
  Book.build({
    name: 'Through the Storm'
  }).save().on('success', function(book) {
    console.log('Book saved');
    Author.build({
      name: 'Lynne Spears',
      biography: 'Author and mother of Britney'
    }).save().on('success', function(record) {
      console.log('Author saved.');
      record.setBooks([book]);
      record.save().on('success', function() {
        console.log('Author & Book Relation created');
      });
    });
  }).on('failure', function(error) {
    console.log('Could not save book');
  });
}).on('failure', function(error) {
  console.log('Failed to sync database');
});
```

为了确保所有的实例都设置正确，需要在等待 book 成功保存到数据库之后再创建 author。同样，在 author 成功保存到数据库之后，book 才能被添加到 author 上。这样确保了 Sequelize 能够取到 author 和 book 的 ID，并且建立它们之间的关系。输出如下：

```
Executing: CREATE TABLE IF NOT EXISTS`AuthorsBooks`
           (`BookId` INT ,`AuthorId` INT ,`createdAt` DATETIME NOT NULL,
           `updatedAt` DATETIME NOT NULL,
           PRIMARY KEY (`BookId` ,`AuthorId` )) ENGINE=InnoDB;
Executing: CREATE TABLE IF NOT EXISTS `Authors`
           (`name` VARCHAR(255), `biography` TEXT,
```

```
            `id` INT NOT NULL auto_increment ,`createdAt` DATETIME NOT NULL,
            `updatedAt` DATETIME NOT NULL, PRIMARY KEY (`id` ))
            ENGINE=InnoDB;
Executing: CREATE TABLE IF NOT EXISTS `Books`
           (`name` VARCHAR(255), `id` INT NOT NULL auto_increment ,
            `createdAt` DATETIME NOT NULL, `updatedAt` DATETIME NOT NULL,
            PRIMARY KEY (`id` )) ENGINE=InnoDB;
Executing: CREATE TABLE IF NOT EXISTS `AuthorsBooks`
           (`BookId` INT , `AuthorId` INT , `createdAt` DATETIME NOT NULL,
            `updatedAt` DATETIME NOT NULL,
            PRIMARY KEY (`BookId` , `AuthorId` )) ENGINE=InnoDB;
Executing: INSERT INTO `Books` (`name` ,`id` ,`createdAt` ,`updatedAt` )
           VALUES ('Through the Storm',NULL,'2011-12-01 20:51:59',
                      '2011-12-01 20:51:59');
Book saved
Executing: INSERT INTO `Authors` (`name` ,`biography` ,`id` ,`createdAt` ,
           `up datedAt` )
           VALUES ('Lynne Spears','Author and mother of Britney',
                      NULL,'2011-12-01 20:51:59','2011-12-01 20:51:59');
Author saved.
Executing: UPDATE `Authors` SET `name` =`Lynne Spears` ,
            `biography` =`Author and mother of Britney` ,`id` =3,
            `createdAt` ='2011-12-01 20:51:59',
            `updatedAt` ='2011-12-01 20:51:59' WHERE `id` =3
Author & Book Relation created
Executing: SELECT * FROM `AuthorsBooks` WHERE `AuthorId` =3;
Executing: INSERT INTO `AuthorsBooks` (`AuthorId` ,`BookId` ,`createdAt` ,
           `u pdatedAt')
           VALUES (3,3,'2011-12-01 20:51:59','2011-12-01 20:51:59');
```

6.2.2 PostgreSQL

PostgreSQL 是面向对象的 RDBMS，诞生于加州大学伯克利分校。其前身是 Ingres 数据库，Michael Stonebraker 教授是该项目的发起者及领头人。从 1985 年到 1993 年，Postres 团队发布了该软件的 4 个版本。在项目接近尾声时，越来越多的用户开始支持此项目，同时提出了大量新功能需求，也使开发团队面临着巨大压力。在伯克利团队开发之后，开源社区的开发者接管了该项目，把原来的 QUEL 语言解析器改为 SQL 语言解析器，并将项目改名为 PostgreSQL。自从 1997 年 PostgreSQL 发布了第一个版本 PostgreSQL 6.0，该数据库系统就作为功能强大的发行版赢得了良好的声誉，对于有 Oracle 背景的用户来说尤其方便好用。

1. 安装

可用于产品线的 PostgreSQL 版本（已经被 Yammer.com 这样的大型网站使用）可以从 npm 资源库下载，如下：

```
npm install pg
```

这需要先安装 pg_config，你可以通过 libpq-dev 包找到它。

2. 选择

例 6-30 假设你已经创建了一个叫做 upandrunning 的数据库并且授予了 dev 用户（密码也是 dev）权限。

例 6-30　从 PostgreSQL 选出数据

```
var pg = require('pg');

var connectionString = "pg://dev:dev@localhost:5432/upandrunning";
pg.connect(connectionString, function(err, client) {
  if (err) {
    console.log( err );
  } else {
    var sqlStmt = "SELECT username, firstname, lastname FROM users";
    client.query( sqlStmt, null, function(err, result) {
      if ( err ) {
        console.log(err);
      } else {
        console.log(result);
      }
      pg.end();
    });
  }
});
```

输出是：

```
{ rows:
   [ { username: 'bshilbo',
       firstname: 'Bilbo',
       lastname: 'Shilbo' } ] }
```

这与 MySQL 驱动的链式调用方法有很大区别。使用 PostgreSQL 的时候，它会要你直接编写 SQL 查询语句。

如上述例子所示，调用 end() 函数会关闭连接，结束 Node 事件循环。

3. 插入、更新和删除

当手工输入 SQL 查询语句时，你可能倾向于直接把数据值通过字符串连接操作扔在代码里，但聪明的程序员会寻求方法来保护自己不被 SQL 注入攻击。pg 库接受参数形式的查询，这样就能够用上来自外部资源（比如网页的表单）里的值。例 6-31 演示了如何插入，例 6-32 和例 6-33 示范的是更新和删除。

例 6-31　插入到 PostgreSQL

```
var pg = require('pg');

var connectionString = "pg://dev:dev@localhost:5432/upandrunning";
pg.connect(connectionString, function(err, client) {
  if (err) {
    console.log( err );
  } else {
    var sqlStmt   = "INSERT INTO users( username, firstname, lastname ) ";
        sqlStmt  += "VALUES ( $1, $2, $3)";
    var sqlParams = ['jdoe', 'John', 'Doe'];
    var query = client.query( sqlStmt, sqlParams, function(err, result) {
      if ( err ) {
        console.log(err);
      } else {
        console.log(result);
      }
      pg.end();
    });
  }
});
```

输出是：

```
{ rows: [], command: 'INSERT', rowCount: 1, oid: 0 }
```

query 命令以 SQL 语句作为第一个参数，第二个参数是一个数据值的数组。MySQL 驱动使用问号作为参数值替换标记，而 PostgreSQL 使用的是序号参数。能够为参数排号有利于你更好地控制数值的组织方式。

例 6-32　在 PostgreSQL 中更新数据

```
var pg = require('pg');

var connectionString = "pg://dev:dev@localhost:5432/upandrunning";
pg.connect(connectionString, function(err, client) {
  if (err) {
    console.log( err );
  } else {
    var sqlStmt   = "UPDATE users "
                  + "SET firstname = $1 "
                  + "WHERE username = $2";
    var sqlParams = ['jane', 'jdoe'];
    var query = client.query( sqlStmt, sqlParams, function(err, result) {
      if ( err ) {
        console.log(err);
      } else {
        console.log(result);
      }
      pg.end();
    });
  }
});
```

例 6-33　从 PostgreSQL 中删除数据

```
var pg = require('pg');

var connectionString = "pg://dev:dev@localhost:5432/upandrunning";
pg.connect(connectionString, function(err, client) {
  if (err) {
    console.log( err );
  } else {
    var sqlStmt   = "DELETE FROM users WHERE username = $1";
    var sqlParams = ['jdoe'];
    var query = client.query( sqlStmt, sqlParams, function(err, result) {
      if ( err ) {
        console.log(err);
      } else {
        console.log(result);
      }
      pg.end();
    });
  }
});
```

6.3　连接池

生产环境通常由多种资源组成：Web 服务器、缓存服务器和数据库服务器。数据库通常是部署在 Web 服务器之外的独立机器上，这使得面向公众的网站不必重新配置和修改复杂的数据库集群就可以垂直增长。因此应用开发程序员需要留心访问这些资源时的性能实现情况，以及这些访问开销会如何影响网站的表现。

连接池在 Web 开发中是非常重要的概念，因为建立一个数据库连接的开销相对来说还是很大的。为每个请求创建一个甚至多个连接会对高流量网站造成不必要的额外负担，也会导致性能下降。解决方案是在内部缓存池里维护数据库连接，当某连接不再需要时，它会被放回连接池里，这样就能立刻为下一个进入的请求服务了。

许多数据库驱动提供了连接池功能，但该模式违反了 Node 的“一个模块，一个功能”的理念。所以，Node 开发者在数据层之上应使用通用的连接池（`generic-pool`）模块来进行数据库连接服务（例 6-34）。`generic-pool` 模块会重用已有的连接，尽可能地防止因为创建新的数据库连接而带来的开销，而且这个模块可以用在任何数据库上。

例 6-34　使用 node-db 的连接池

```
var mysql = require( 'db-mysql' );
var poolModule = require('generic-pool');

var connectParams = {
  'hostname': 'localhost',
```

```
    'user': 'dev',
    'password': 'dev',
    'database': 'zborowski'
  }

  var pool = poolModule.Pool({
    name : 'mysql',
    create : function(callback) {
      var db = new mysql.Database( connectParams );
      db.connect(function(error) {
        callback(error, db);
      });
    },
    destroy : function(client) { client.disconnect(); },
    max     : 10,
    idleTimeoutMillis : 3000,
    log : true
  });

  pool.acquire(function(error, client) {
    if ( error ) return console.log("Failed to connect");

    client.query()
      .select(['id', 'user_login'])
      .from('wp_users')
      .execute(function(error, rows, columns) {
        if ( error ) {
          console.log("Error on query");
        } else {
          console.log(rows);
        }
        pool.release(client);
      });
  });
```

输出为：

```
pool mysql - dispense() clients=1 available=0
pool mysql - dispense() - creating obj - count=1
[ { id: 1, user_login: 'mwilson' } ]
pool mysql - timeout: 1319413992199
pool mysql - dispense() clients=0 available=1
pool mysql - availableObjects.length=1
pool mysql - availableObjects.length=1
pool mysql - removeIdle() destroying obj - now:1319413992211
timeout:1319413992199
pool mysql - removeIdle() all objects removed
```

连接池通过神奇的创建（create）和销毁（destroy）函数来工作。当客户尝试获取一个连接时，如果没有已经打开的连接，连接池会调用创建函数。如果一个连接闲置太久了（由 idleTimeoutMillis 属性来指定空闲间隔，以毫秒来计算），它会被销毁并且释放内存资源。

Node 连接池的优雅之处在于，它可以表示任何持久化的资源。选择数据库可谓得天独厚，同时你也可以轻松地写些命令来维护与外界资源（如会话缓存，甚至是硬件接口）的连接。

6.4 消息队列协议

之前，我们举了邮递员的例子来描述 Node 的事件循环架构。如果邮递员碰到哪家关了门，他就无法继续投递信件了。想象一下，如果有一名好心的老门卫能够把门打开，让邮递员通过呢？但是门卫已经上了年纪，且因为服务多年而身体虚弱，他需要多花点时间才能清理道路，因此这段时间内邮递员暂时无法继续投递信件。

这就类似堵塞的进程，但这种状况不会一直持续。最终，门卫会把门打开，然后邮递员又能继续他的业务了。如果邮递员到达的每一个屋子都有类似的开门进程，会把整个通道都拖慢。在 Node 程序里，这类堵塞将严重降低系统性能。

在计算机领域，造成类似情况的原因很多，可能是因为在注册过程中需要发送用户邮件，需要对用户输入进行大量的数学运算，或者是某个任务需要花费的时间超过了用户期望的等待时间，等等。Node 的事件驱动设计可用来应对大多数情况，它采用的是异步函数和回调的方法。但是如果一个事件特别“重”的话，就不应该放在 Node 内部处理。Node 应该只负责快速运算和处理返回的结果。

以一个普通的用户注册流程为例。当用户自己注册时，应用程序会在数据库中保存一条新的记录，并发送邮件给该用户。它也许还会记录下注册过程中的一些统计数据，比如整个过程包括了几个步骤、花费了多少时间。如果用户刚在你的网页上点击提交按钮，系统就马上处理那么多的操作，其实并没有太大意义。比如，发送邮件的流程也许需要花费几秒钟（如果你运气不佳，要花上几分钟）来完成，数据库调用可以等到用户受到欢迎之后再进行操作，统计数据可以从程序的主逻辑独立出去处理。这样的情况下，你可以选择生成一条消息，来通知程序的其他部分有新用户注册了，这样的程序也可能是完全运行在另外一台服务器上的。这就是我们所称的*发布 – 订阅模型*（publish-subscribe pattern）。

再假设你有一个集群的机器运行了 Node.js 程序。当一台新机器要加入到集群的时候，它发出一条信息来请求配置信息。配置服务器返回的信息包含了新机器整合到集群中所需要的配置信息列表，这称为*请求 – 回复模型*（request-reply pattern）。

消息队列允许程序员发布事件然后继续其他操作，通过进程间通信频道，提高了并发处理的效率，并实现了更高的扩展性。

RabbitMQ

RabbitMQ 是一个消息代理，支持高级消息队列协议（AMQP）。它适用的情景有跨服务器的数据交换和同一台服务器上的跨进程通信。RabbitMQ 使用 Erlang 语言编写，能够提供集群的高可用性，并且很容易安装和使用。

1. 安装 RabbitMQ

如果你使用 Linux 系统，大部分发行版都提供了 RabbitMQ 的安装包。你也可以从 http://www.rabbitmq.com 下载此软件，然后从源代码编译。

一旦安装好了 RabbitMQ，并启动起来，就可以使用 npm 来获得 Node 的 AMQP 驱动：

```
npm install amqp
```

2. 发布与订阅

RabbitMQ 使用标准的 AMQP 协议进行通信。AMQP 源于金融服务行业，在金融领域消息的可靠性关系重大。AMQP 提供了对厂商中立的抽象规范，可以提供通用的（不只针对金融行业的）消息中间件服务，并且旨在解决不同类型系统间通信的问题。AMQP 与 E-mail 的概念很像：E-mail 消息有其头信息和格式的规范，但内容可以是任何格式，文本、图片或视频都可以；两个公司之间不需要运行同一款 E-mail 服务器就能通信。AMQP 还可以在不同平台间通信。比如，用 PHP 编写的发布者可以给用 JavaScript 编写的消费者发送消息。

例 6-35　AMQP/RabbitMQ 使用方法

```
var connection = require('amqp').createConnection();

connection.on('ready', function() {
  console.log('Connected to ' + connection.serverProperties.product);
  var e = connection.exchange('up-and-running');

  var q = connection.queue('up-and-running-queue');

  q.on('queueDeclareOk', function(args) {
    console.log('Queue opened');
    q.bind(e, '#');

    q.on('queueBindOk', function() {
      console.log('Queue bound');

      q.on('basicConsumeOk', function() {
        console.log("Consumer has subscribed, publishing message.");
        e.publish('routingKey', {hello:'world'});
      });
    });
```

```
        q.subscribe(function(msg) {
          console.log('Message received:');
          console.log(msg);
          connection.end();
        });
      });
    });
```

输出为：

```
Connected to RabbitMQ
Queue opened
Queue bound
Consumer has subscribed, publishing message.
Message received:
{ hello: 'world' }
```

`createConnection` 命令建立了一个到 RabbitMQ 消息代理的连接，默认情况（依照 AMQP 协议）是 localhost 的 5672 端口。如果需要，这个命令可以被重载，如：

```
createConnection({host: 'dev.mycompany.com', port: 5555})
```

接下来定义了 queue 和 exchange。这一步并不是严格要求的，因为 AMQP 代理会被要求提供一个默认的 exchange。但是通过指定 `up-and-running` 作为 exchange 的名字，你可以让程序与运行在同一台服务器上的其他 exchange 隔离开。exchange 是负责接收消息并把它们传递给绑定的队列的实体。

队列自己并不会做任何操作，它必须绑定到某个 exchange 之后才能进行其他操作。`q.bind(e, '#')` 命令告诉 AMQP 把名为 `up-and-running-queue` 的队列添加到名为 `up-and-running` 的 exchange 上，并且让 exchange 监听所有传给它的消息（通过 `'#'` 参数）。你可以很方便地把 # 改为其他关键字来过滤消息。

一旦声明好了 queue 和 exchange，我们就设置了 `basicConsumeOk` 的事件监听，当客户端订阅了此队列之后，AMQP 库会触发这个事件。而后 Node 会发布一条 hello world 消息，以及用来过滤的关键词 `routingKey` 给 exchange。在这个例子里，过滤的关键词是什么并没有关系，因为队列绑定了所有内容（通过 `bind('#')` 命令）。但 AMQP 的中心思想是发布者永远不知道哪些订阅者连接了，所以需要有一个作为路由的关键词备用。

最后，发送了 `subscribe` 命令。回调函数是作为参数传入的，并且每次符合条件的消息传给 exchange 并通过 queue 传输之后，都会调用它。在本例中，回调函数会导致程序退出，这对于演示的目的来说足够了。但在真实的应用中，你不太可能这样做。当 `subscribe` 命令成功执行后，AMQP 会分发 `basicConsumeOk` 事件，这会触发 hello world 消息的发布，然后中断演示程序。

3. 工作队列

如果长时间运行的任务超出了用户的容忍度（比如等待一个网页加载时），或者是该任务会堵塞整个程序，使用队列就很合适。使用 RabbitMQ，是否可以把任务分散到多个工作进程中，并确保所有任务都能完成呢？即使第一个工作进程在处理它们的时候中途死掉也行吗？（例 6-36）

例 6-36　用 AMQP 发布长时间运行任务

```
var connection = require('amqp').createConnection();
var count = 0;

connection.on('ready', function() {
  console.log('Connected to ' + connection.serverProperties.product);
  var e = connection.exchange('up-and-running');

  var q = connection.queue('up-and-running-queue');

  q.on('queueDeclareOk', function(args) {
    console.log('Queue opened');
    q.bind(e, '#');

    q.on('queueBindOk', function() {
      console.log('Queue bound');

      setInterval(function(){
        console.log('Publishing message #' + ++count);
        e.publish('routingKey', {count:count});
      }, 1000);
    });
  });
});
```

这个例子是上一节的简单发布 – 订阅例子的修改版。但它只是一个发布者，所以移除了订阅相关的事件监听器。代替它的是一个定时器，用来每隔 1000 毫秒（即每秒）往队列发布一条消息。该消息包含了一个 `count` 变量，表示每次发布增加的次数。这段代码可以用来实现一个简单的工作者应用。例 6-37 示范了如何写相应的客户端。

例 6-37　用 AMQP 处理长时间运行任务

```
var connection = require('amqp').createConnection();

function sleep(milliseconds)
{
  var start = new Date().getTime();
  while (new Date().getTime() < start + milliseconds);
}

connection.on('ready', function() {
```

```
    console.log('Connected to ' + connection.serverProperties.product);

    var e = connection.exchange('up-and-running');
    var q = connection.queue('up-and-running-queue');

    q.on('queueDeclareOk', function(args) {
    q.bind(e,'#');

    q.subscribe({ack:true},function(msg) {
      console.log('Message received:');
      console.log(msg.count);
      sleep(5000);
      console.log('Processed. Waiting for next message.');
      q.shift();
      });
    });
  });
```

客户端从队列获取消息再处理它（在这个例子里是睡眠 5 秒），然后从队列获取下一条消息，并不断重复。虽然 Node 里并没有 sleep 函数，但你可以用死循环来代替它，如例子所示。

这里有个问题。回想一下，前面的发布者是每隔 1 秒发送一条消息到队列的，但因为客户端要花费 5 秒才能处理一条消息，它会很快就落后于发布者。解决方法呢？打开另外一个窗口并运行第二个客户端，现在消息处理的速度已经翻倍了。但这样依然不够快，还不能跟上发布者生产的数量。通过增加客户端可以进一步分散负载，并且避免让未处理的消息落在后面，这种部署就称为*工作队列*。

工作队列的工作原理是发布的消息在连接到队列的客户端间循环触发。`subscribe` 命令的 `{ack:true}` 参数是通知 AMQP 等待用户确认，看该消息是否已经处理完成。此反馈确认由 `shift` 方法提供，该方法在应答过后把消息从队列中移除，同时将其从服务里拿掉。这样，如果一个工作进程在处理某个消息的过程中死掉了，RabbitMQ 代理会把消息发给下一个可用的客户端。这里不需要使用超时，只要客户端是连接上的，消息就会从工作流中移除。只有当客户端没有发送反馈就断开了，该消息才会被发送到下一个客户端。

> 一个常见的“疑难杂症”是开发者常常忘记调用 `q.shift()` 命令。如果你忘记这个操作，程序依然会正常运行。但当某个客户端断开的时候，服务器会把该客户端处理过的所有消息都重新放到队列中。
> 另一个副作用是，RabbitMQ 使用的内存会不停地上升。这是因为，即使消息已经从队列的活跃状态中移除，它们依然会保存在内存里，直到等来客户端的反馈结果并被客户端移除。

第 7 章

重要的外部模块

虽然 Node 提供的核心 API 很强大，但其中很多功能还是过于底层。正如许多基于 Ruby 的网站会使用 Rails 和 Sinatra 开发，而不是使用自己编写的 Ruby 代码一样，Node 社区构建了许多高级的功能库，大大地方便了开发。虽然这些模块从技术上讲并不是 Node 本身，但它们对完成工作非常重要，而且其中许多库自身就是成熟的项目。本章将介绍 Node 社区提供的几个最流行及最有用的模块。

7.1 Express

Express（Node 的 MVC 框架）也许是使用最广泛的 Node 模块了，它吸取了 Ruby 的 Sinatra 框架的精髓，并提供了许多功能，使得用 Node 构建网站变得非常简单。

7.1.1 一个简单的Express应用

Express 通过路由定义的页面处理器来工作。路由可以是一个简单的路径，也可以比较复杂。处理器可以简单地输出 Hello, world，也可以复杂得像一个与数据库交互的完整页面渲染系统。需要先运行 `npm install express` 来安装 Express，然后才能开始使用它。例 7-1 演示了如何用 Express 创建一个简单的应用。

例 7-1　创建一个简单的 Express 应用

```
var express = require('express');

var app = express.createServer();
```

```
app.get('/', function(req, res) {
  res.send('hello world');
});

app.listen(9001);
```

从创建服务器的角度来看，这段代码与 http 模块非常相似，但有几点更加直观。首先，app.get() 为特定的路由（例子中为 '/'）创建了响应函数。与一般的 http 服务器不同的是，Express 不是为一般的请求提供监听器，而是针对特定的 HTTP 动作提供监听器。所以 get() 只会响应 GET 请求，put() 只处理 PUT 请求，等等。配合上我们指定的路由，你马上可以拥有更为强大的功能。一个典型的 Express 程序会指定一系列表达式，Express 会为每个进入的请求对所有的表达式按序进行路由匹配，然后执行第一个匹配的表达式对应的代码。

> 我们还可以利用 next() 函数，让 Express 在特定的情况下跳过表达式，这会在本章后面进行讨论。

接下来看看我们是如何响应请求的。我们依然使用同 http 中一样的 response 对象，但 Express 增加了 send() 方法。因此，我们不需要手动提供 HTTP 头或者是调用 end() 方法，send() 方法会处理好如何发送 HTTP 头等操作，并会自动包含 end() 调用。

这里需要了解的是，Express 是把 http 的基础功能封装起来，并通过提供众多的功能来丰富它，使得创建真正的应用非常便捷。你不需要在每次处理 HTTP 请求的时候自己编写代码来处理路由逻辑，Express 都为你处理好了。

7.1.2 在Express中设置路由

路由是 Express 的一个核心概念，也是 Express 非常有用的原因之一。正如上一小节中介绍的那样，路由通过实现同名的方法（如 get()、post()）来支持一个 HTTP 动作。路由包含了一个简单的字符串或者一个正则表达式，并且可以包含变量声明、通配符及可选关键字标记。让我们看几个例子，先从例 7-2 开始。

例 7-2　通过变量和可选标记选择路由

```
var express = require('express');
var app = express.createServer();

app.get('/:id?', function(req, res) {
  if(req.params.id) {
    res.send(req.params.id);
  } else {
```

```
    res.send('oh hai');
  }
});

app.listen(9001);
```

本例演示了如何在一个路由中包含一个可选的变量 id。Express 并不关心变量的取名，但之后能够在回调函数中使用它。在 Express 路由中，我们使用冒号（:）来标记想要使用的变量，那么在 URL 中传递的字符串就会被捕获并保存在该变量中。Express 中的所有路由最终都会转变成正则表达式来处理（稍后会做更多介绍），并进行分词[1]操作以便于应用代码的使用。[2]正则表达式用来匹配路由中下一个已知的词。注意，这个变量其实是可选的。如果你运行此程序，然后访问 http://localhost:9001，就只会得到"oh hai"的响应，因为你没有在端口后面用 / 带上路由的可选变量部分。如果随意带上点什么内容（只要不包含另外一个 / 在里面），就将在响应内容中得到一样的内容，因为匹配了 id 关键字的内容会被保存在 req.params.id 中。

Express 路由总是将 / 视作一个标记，而同时又会把请求末尾的 / 当做可选项，所以我们提供的路由 /:id? 会匹配上 localhost、localhost/、localhost/tom 和 localhost/tom/，但不包括 localhost/tom/tom。

路由中也可以使用通配符，如例 7-3 所示，(*) 会匹配所有的内容，直到下一个标记出现（非贪心的正则匹配）。

例 7-3　在路由中使用通配符

```
app.get('/a*', function(req,res) {
  res.send('a');
  // 匹配 /afoo /a.bar /a/qux 等
});

app.get('/b*/c*d', function(req,res) {
  res.send('b');
  // 匹配 /b/cd /b/cfood /b//c/d/ 等
  // 不匹配 /b/c/d/foo
});

app.get('*', function(req, res) {
  res.send('*');
  // 匹配 /a /c /b/cd /b/c/d /b/c/d/foo
  // 不匹配 /afoo /bfoo/cbard
});
```

注 1：分词（tokenize）指的是把一个文本字符串按块或按词切分成一个个标记的过程。

注 2：这个功能实际上是 Express 的一个子模块 router 完成的。你可以参考 router 模块的源代码来了解更多关于路由正则表达式的细节。

当使用通配符来构建路由时，两个通配符之间的标记必须匹配，除非它是可选的。通配符通常用在包含（.）的文件名中。还需要注意的是，与许多其他正则表达式语言不同，* 表示的不是零个以上字符，它表示的是一个以上字符。一个斜杠（/）在匹配通配符的时候可以认为是一个字符。

另外需要注意的是，路由是按顺序执行的。当多个路由同时匹配上提供的 URL 时，只有第一个匹配的路由会执行相关的动作，也就是说，如何安排路由的顺序是很重要的。在前面的例子里，即便通配符能够匹配所有的 URL，它也只能捕获前面的路由未能匹配的 URL。

你还可以用正则表达式来定义路由（例 7-4）。如果使用了这个方法，路由不会对正则匹配的内容做进一步处理。所以当你想从 URL 中提取变量时，需要用正则的语法进行处理。

例 7-4　用正则表达式来定义路由

```
var express = require('express');
var app = express.createServer();

app.get(/\/(\d+)/, function(req, res) {
  res.send(req.params[0]);
});

app.listen(9001);
```

在这个例子里，正则表达式只会匹配以数字开头的 URL（\d 表示匹配任意的数字，+ 允许匹配 1 个或多个）。这表示 / 不会被匹配，而 /12 则会被匹配。其实，正则匹配调用的是 RegExp.match() 方法，它会在一个较长的字符串中寻找匹配的部分内容，所以 /12abc 也会被匹配。如果你想确保正则表达式匹配完整的路由，就需要在表达式的末尾添加 $ 标记，如 /\/(\d+)$/。因为 $ 会检查是否为行尾，所以这个表达式只有结束了才会被匹配。

也许你想让 Express 的默认行为忽略 URL 结尾的 /，那么只要在所有的字符串末尾用 \/?$ 替换 $ 就可以了。

注意看我们在例 7-4 中是如何访问正则表达式中提取的变量的。当你在路由中使用正则表达式时，可以通过 req.params 数组来访问提取的变量。当 router 模块把你提供的路由处理成正则表达式时，这也同样有效，只不过你可能更希望通过变量名来访问罢了（如之前的例子所示）。你也可以在路由的正则匹配时指定变量的命名，如例子 7-5 所示。

例 7-5　用正则表达式的同时指定变量类型

```
var express = require('express');
var app = express.createServer();

app.get('/:id(\\d+)', function(req, res) {
  res.send(req.params[0]);
});

app.listen(9001);
```

在这个例子中，通过设置路由只匹配数字（正则表达式 `\d+`）来限定参数 `id` 只能是数字。提取的变量依然可以通过 `req.params.id` 得到，但前提是能够被该正则匹配。因为正则表达式非常灵活，所以你可以用此技术来捕捉或限制 URL 该匹配的情况，并能够方便地使用命名变量。注意，当你在 JavaScript 字符串里输入反斜杠（\）时，需要对它进行转义。（但例 7-4 中的情况不需要进行转义操作，因为那是直接在正则表达式中使用，而不是在字符串中。）

有时候，你会希望同一个 URL 在不同的情景下匹配上多个路由。我们已经看到了路由定义的顺序会决定哪个路由被选中使用。但是，当某些条件不满足的时候（例 7-6），依然有办法可以把控制权传给下一个路由，这在许多情况下会很有用。

例 7-6　把控制权传给下一个路由

```
app.get('/users/:id', function(req, res, next){
  var id = req.params.id;

  if (checkPermission(id)) {
    // 显示个人页面
  } else {
    next();
  }
});

app.get('/users/:id', function(req, res){
  // 显示公共页面
});
```

我们对路由的处理函数增加了一个新的参数，`next` 参数会通知路由中间件去调用下一个路由（稍后我们会详细介绍中间件）。这个参数总是会传递给回调函数的，只不过我们在这个例子里是第一次为它命名并使用它。在这个例子里，我们通过检查 `id` 来查看用户是否有权限查看该页面的私有版本。如果没有权限，就把他带到下一个显示公共版本的路由去。

这还能与 `app.all()` 方法很好地配合，它表示所有的 HTTP 动作都进行处理。如例 7-7 演示的内容，我们可以同时捕捉各种 HTTP 动作及路由，添加一些处理逻辑，然后把控制权交给更加具体的路由。

例 7-7　使用 app.all() 来处理不同的 HTTP 动作和路由，然后交回控制权

```
var express = require('express');

var app = express.createServer();

var users = [{ name: 'tj' }, { name: tom }];

app.all('/user/:id/:op?', function(req, res, next){
  req.user = users[req.params.id];

  if (req.user) {
    next();
  } else {
    next(new Error('Cannot find user with ID: ' + req.params.id));
  }
});

app.get('/user/:id', function(req, res){
  res.send('Viewing ' + req.user.name);
});

app.get('/user/:id/edit', function(req, res){
  res.send('Editing ' + req.user.name);
});

app.put('/user/:id', function(req, res){
  res.send('Updating ' + req.user.name);
});

app.get('*', function(req, res){
  res.send('Danger, Will Robinson!', 404);
});

app.listen(3000);
```

这个例子与例 7-6 相似，我们在传递控制权之前先检查该用户是否存在。但是，我们不但对后续的路径进行此操作，同时还对所有的 HTTP 操作都进行了检查。通常只会有一个路由被匹配上，因此这样的写法没有什么区别。但重要的一点是，要注意如何在路由间直接传递数据。

因为中间件拥有 request 对象，所以当 app.all() 方法中添加了 req.user 属性后，后续所有的方法都能访问到它。每当回调函数被触发时，变量 .req 其实只是指向中间件所持有的 request 对象。所以使用中间件的所有函数和路由对 request 对象的任何操作都是可见的。

例 7-8 展示了在特定范围内如何把一个文件扩展名设置为可选或者是必选。在第一个 get() 方法里，:format 参数是可选的（正如 ? 所标记的），所以 Express 会对用户请求的 ID 进行响应，而不会在乎其请求的是哪种格式。它会交给程序员来根

据不同的格式（JSON、XML、文本等）采取相应的处理。

在第二个例子中，`:format` 参数期望匹配的类型是 `json` 或 `xml`。如果没有匹配的类型，即便 `:id` 参数是有效的，请求 book 的操作也不会进行。这让我们在决定处理哪些请求时有了很强的控制权，并且能够确保只有符合的格式才会被处理并生成响应内容。

例 7-8　可选或必填的路由扩张项

```
var express = require('express');
var app = express.createServer();

app.get('/users/:id.:format?', function(req, res) {
  res.send(req.params.id + "<br/>" + req.params.format);
  // 会响应：
  // /users/15
  // /users/15.xml
  // /users/15.json
});

app.get('/books/:id.:format((json|xml))', function(req, res) {
  res.send(req.params.id + "<br/>" + req.params.format);
  // 会响应：
  // /books/7.json
  // /books/7.xml
  // 但不会处理：
  // /books/7
  // /books/7.txt
});

app.listen(8080);
```

7.1.3　处理表单数据

大部分例子都演示了如何使用 GET 动作，其实 Express 是以 Ruby on Rails 的风格来提供 RESTful 的架构的。利用 Web 表单的隐藏项，你可以让表单进行各种操作：PUT（替换数据）、POST（创建数据）、DELETE（删除数据）和 GET（获取数据）。我们来看一下例 7-9。

例 7-9　用 Express 处理表单

```
var express = require('express');
var app = express.createServer();

app.use(express.limit('1mb'));
app.use(express.bodyParser());
app.use(express.methodOverride());

app.get('/', function(req, res) {
```

```
  res.send('<form method="post" action="/">' +
           '<input type="hidden" name="_method" value="put" />' +
           'Your Name: <input type="text" name="username" />' +
           '<input type="submit" />' +
           '</form>');
});

app.put('/', function(req, res) {
  res.send('Welcome, ' + req.body.username);
});

app.listen(8080);
```

这个简单的例子演示了如何使用表单。首先，创建一个 Express 应用并配置使用 bodyParser() 和 methodOverride() 方法。bodyParser() 方法会解析从 Web 浏览器发送来的请求正文，并把表单变量转换成 Express 使用的对象。methodOverride() 方法允许表单提交隐藏的 _method 变量，并把 GET 方法替换掉，然后调用相应的 RESTful 方法类型。

express.limit() 方法指示 Express 把请求正文的大小限制在 1MB 以内。这是一个很重要的安全考虑，因为如果没有这个设置，外界就可以给应用发送一个超大的内容并让 bodyParser() 来处理，这样就很容易进行一个拒绝服务（denial-of-service，Dos）攻击。

> 请确保在 bodyParser() 之后再调用 methodOverride() 方法，否则，当 Express 检查是否要处理 GET 方法或者其他命令时，表单变量就无法被处理。

7.1.4 模板引擎

显然，直接在应用代码里继续手写 HTML 代码并不明智。对初学者来说，它既不可读也不可维护，但更为重要的原因是，把代码逻辑与表现层标记混合在一起是一个糟糕的做法。模板引擎允许程序员把精力集中在如何把信息呈现给用户上（通常以不同的格式，如显示器或手机格式），并把嵌入专用数据的过程从处理流程中分离开来。

Express 是最小化的集合，因此并没有内置模板引擎，而是由社区开发的模块来竞争。一些流行的引擎有 Haml、Jade、Embedded Javascript（EJ）、CoffeeKup（基于 CoffeeScript 的引擎）和 jQuery 模板。

例 7-10 中，应用的功能是渲染一个简单的 Jade 模板。

例 7-10　Express 中使用简单的 Jade 模板

```
var express = require('express');
var app = express.createServer();

app.get('/', function(req, res) {
  res.render('index.jade', { pageTitle: 'Jade Example', layout: false });
});

app.listen(8080);
```

运行此例子，需要先安装 Jade 模板引擎：

```
npm install jade
```

首先注意的是，代码里并不需要引用 Jade 库，Express 会解析视图使用的模板文件名，并根据扩展名来确定该用哪个模板引擎（本例子中是 index.jade 的 jade）。因此，我们可以在一个项目中混合使用多种模板引擎。比如，项目并不会限制你只使用 Jade 或只使用 CoffeeKup，而是可以同时使用两个。

这个例子传了两个参数给 render 函数，第一个是用来显示视图的名字，第二个包含了配置项以及渲染需要的变量。我们稍后再回过头来讨论文件名参数。在本例里，我们传给视图两个变量：`pageTitle` 和 `layout`。`layout` 变量在这个例子中比较特别，因为它被设置为 `false`，意思是让 Jade 模板引擎在渲染 index.jade 的时候，不要采用 layout 模板文件（这一点后面会详细讲）。

`pageTitle` 是一个本地变量，并会被视图内容所使用。它代表了模板的使用原理：在 index.jade 文件中指定的 HTML 内容中，有一个叫做 `pageTitle` 的占位符，Jade 会把我们传递的值替换上去。

第一个参数 index.jade 文件需要放在 views 文件夹下（/views/index.jade），如例 7-11 所示。

例 7-11　Express 使用的简单 Jade 文件

```
!!! 5
html(lang="en")
  head
    title =pageTitle
  body
    h1 Hello, World
    p This is an example of Jade.
```

在 Jade 把我们提供的 `pageTitle` 值填入页面后，渲染得到的效果如下：

```
<!DOCTYPE html>
<html lang="en">
```

```
  <head>
    <title>Jade Example</title>
  </head>

  <body>
    <h1>Hello, World</h1>
    <p>This is an example of Jade.</p>
  </body>
</html>
```

Jade 模板引擎通过把使用标记降低到最少，使得页面尽量简洁。你也许已经熟悉 HTML 里的各种标签的结束记号了，Jade 取代它们的方法是根据缩近来确定页面的结构，所以得到的文件非常干净且容易阅读。

第 1 行 `!!! 5` 表示内容类型为 HTML5，它会在输出结构中声明 HTML5 的 doctype。Jade 支持的默认文件类型有 `5`、`xml`、`default`（XHTML 1.0 Transitional）、`transitional`（默认类型）、`strict`、`frameset`、`1.1`、`basic` 和 `mobile`。同时，你也可以指定自己的格式，比如 `doctype html PUBLIC "-//W3C//DATA XHTML Custom 1.10a//DE"`。

我们来看一下 Jade 输入的第 4 行的 `title` 标记，Jade 会把字符串 `=pageTitle` 解析为“把变量 `pageTitle` 的内容插入到此处”。在输出结果中，内容变成了 `Jade Example`，这正是前文代码提供的值。

我们已经介绍过，还有许多其他的模板选择，这些模板的功能也和 Jade 相差无几，只不过各有各的语法和约束。

布局与子视图

布局能让你的网站实现视图中常用元素的共享，把内容与数据进一步分离开。通过把布局中的部件标准化，如导航、页眉和页脚，你可以把开发精力集中在网页视图的实际内容上。

例 7-12 把刚才讨论的视图引擎例子放在一个“真实”的网站上了。

例 7-12　定义 Express 中的全局模板引擎

```
var express = require('express');
var app = express.createServer();

app.set('view engine', 'jade');

app.get('/', function(req, res) {
  res.render('battlestar')
});
```

这个例子中新增加了 `set` 命令的 view engine 参数，现在 Express 把 Jade 引擎作为默认的模板引擎了，但依然允许在 render 调用的时候修改掉。

render 函数与之前的大不一样，因为已经设置了 Jade 作为默认模板引擎，例子中无需指定文件全名，`battlestar` 实际代表了 /views/battlestar.jade。如例 7-13 所示，Express 会使用放在 views/layout.jade 中的文件作为布局，因此不需要再像例 7-10 中那样把 `layout` 设置为 `false`。

例 7-13　Express 中的 `layout` 文件

```
html
  body
    h1 Battlestar Galactica Fan Page
    != body
```

layout 文件与之前创建的视图文件很像，但是在这个例子中有一个特殊的 `body` 变量。现在解释一下 `!= body` 这行的意思，注意不要和文件顶部的 `body` 关键词混淆。第二个 `body` 并不是从应用代码传递过来的一个变量名，那么它是从哪里来的呢？

当 `layout` 选项在 Express 中设置为 `true` 时（默认），`render` 方法会把第一个参数的内容解析出来，然后把渲染好的输出以变量 `body` 传递给 layout，battlestart.jade 文件如例 7-14 所示。

例 7-14　Express 中的 Jade 子视图

```
p Welcome to the fan page.
```

这里之所以称为子视图，是因为它并没有包含需要生成页面的所有内容，它需要结合 layout 才能变成有用的输出结果。最终在 Web 浏览器的输出如下：

```
<html>
  <body>
    <h1>Battlestar Galactica Fan Page</h1>
    <p>Welcome to the fan page.</p>
  </body>
</html>
```

子视图很有用，因为它使得开发者可以把注意力集中在需要显示的特定内容上，而不需要关注整个网页。这意味着内容不需要绑定在某个网页上，而且也可以作为手机页面的输出、AJAX 调用的结果（页面内刷新），等等。

> 注意不要混淆了变量 `body` 与关键字 `body`，变量 `body` 包含了你的视图的实际内容，而关键字 `body` 是 Web 浏览器使用的一个 HTML 标签。

7.1.5 中间件

在这之前的例子中，有些地方包含了看起来无关的函数 `app.use()`。这个函数调用了 Connect 库，并提供了许多有用的工具，使得添加功能很容易。现在是时候进一步介绍这个神奇的黏合剂（中间件），以及它为什么对开发 Express 程序如此重要了。

“中间件”（middleware）这个名称可能听起来有点像那些爱显摆的程序员喜欢使用的难懂术语，但正如前面章节中提到的，中间件指的是链接两个程序的一个软件，并且通常是更高级的程序间或者更宽广的网络间的软件。在实际工作中，中间件好比你家里或办公室里看得见的电话线，所有电话（应用）都连接到电话线（中间件）上，而后者能把应用与对应底层的网络链接起来。

你的电话也许支持呼叫等待或语音信箱，但不管什么功能，线路工作方式都一样。你可以把语音信箱内置在电话中，或者是通过电信公司（网络）提供，但无论哪种情况，线路本身都乐于为你服务。

Connect 库提供了 Express 使用的中间件功能（见表 7-1）。如图 7-1 所示，Connect 扩展了 Node 的基础 `http` 模块，为它赋予了 `http` 能提供的所有基础服务，然后在此基础上又增加了自己的功能。Express 是从 Connect 继承下来的，同时获得了 `http` 和 `Connect` 的功能。任何添加到 Connect 的模块都会自动被 Express 所使用。Connect 是链接 Express 和网络的中间层，它提供及使用了众多的功能，这些功能也许不能被 Express 直接使用，但可用性都是一样的。最后，因为 Express 本身从 Connect 而来，所以 Connect 的大部分功能都能直接从 Express 中使用，这使得你可以使用 `app.bodyParser()` 这类命令，而不需要调用 `connect.bodyParser()` 之类。

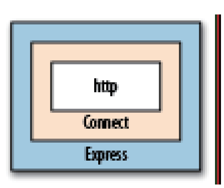


图 7-1：Express 的中间件栈

表7-1：与Connect绑定的中间件

名　称	描　述
`basicAuth`	回调函数接受 `username` 和 `password` 参数，如果该证书允许访问网站，则函数返回值为真
`bodyParser`	解析请求正文的内容
`compiler`	把 .sass 和 .less 文件编译 成 CSS，把 CoffeeScript 文件编译成 JavaScript

（续）

名　　称	描　　述
.cookieParser	解析从浏览器发送过来的请求包头中 cookie 的内容
csrf	通过改变额外的表单变量提供伪造跨域请求（CSRF），依赖 session 和 bodyParser 中间件
directory	打印根路径下的文件夹，可以选择是否显示隐藏文件和图标
errorHandler	捕获应用遇到的错误，提供选择把错误记录到 stderr 或者是其他格式（JSON、纯文本或 HTML）
favicon	通过缓存控制，从内存提供 favicon 文件服务
limit	限制服务器能接受的请求的大小，避免 DoS 攻击
logger	在响应时（默认）或接收请求时，把请求内容记录到标准输出或文件，并且允许使用多种格式。通过改变缓存大小来控制写到磁盘的频率
methodOverride	结合 bodyParser 提供 DELETE、PUT 以及 POST 方法。允许更为明确的路由定义，比如，使用 app.put() 来替代在 app.post() 中检查用户的意图。这个技术能够帮助设计 RESTful 应用。
profiler	一般是放在其他所有中间件的前面，它会记录请求对应的响应时间和内存使用情况
query	解析 query 字符串，然后保存到 req.query 参数中
responseTime	生成响应内容时，把时间（单位是毫秒）填入 X-Response-Time 头
router	提供高级路由功能（详见 7.1.2 节）
session	让多个请求共享用户数据的 session 管理
static	允许从根目录提供静态文件服务。支持部分下载和自定义过期时限
staticCache	为 static 中间件增加缓存层，通过把热门文件放在内存而提高响应速度
vhost	允许在单一机器上以不同的 vhost 提供站点服务

中间件工厂

到目前为止，你也许注意到了中间件包含的功能比起 Express 顺序执行的函数要多一些。JavaScript 的闭包让我们能够在 Node 内部实现工厂模式[3]，并且能为你的网站路由提供上下文处理功能。

Express 的路由功能会在处理环节使用内部的中间件，可以通过重载来添加额外的功能，比如，在 HTML 输出中添加自定义头。我们来看一下例 7-15，看看如何利用中间件工厂来拦截一个页面请求，并且强制进行角色授权验证。

例 7-15　Express 中的中间件工厂

```
var express = require('express');
var app = express.createServer(
  express.cookieParser(),
  express.session({ secret: 'secret key' })
```

注 3：工厂是一个对象，它根据指定的参数创建出其他对象来。而如果采用手工创建对象的方式，会导致出现大量重复或复杂的代码。

```
);

var roleFactory = function(role) {
  return function(req, res, next) {
    if ( req.session.role && req.session.role.indexOf(role) != -1 ) {
      next();
     } else {
      res.send('You are not authenticated.');
    }
  }
};

app.get('/', roleFactory('admin'), function(req, res) {
  res.send('Welcome to Express!');
});

app.get('/auth', function(req, res) {
  req.session.role = 'admin';
  res.send('You have been authenticated.');
});

app.listen(8080);
```

马上就可以测试一下，访问 http://localhost:8080/，你会收到消息“You are not authenticated.”。但是，如果看一下 `'/'` 路由对应的内容，你会发现实际上页面的内容是 `'Welcome to Express!'`。第二个参数 `roleFactory('admin')` 会在页面显示前被调用，并检测到你的 session 中没有 `role` 属性，所以它停止了页面执行并且输出自己的消息。

如果访问 http://localhost:8080/auth 之后，接着访问 http://localhost:8080/，你会得到消息“Welcome to Express!”。在此情景下，/auth URL 往你的 session 的 `role` 属性上添加了 `'admin'` 变量，所以当 `roleFactory` 执行时，它会把执行控制交给 `next()`，也就是 `app.get('/')` 函数。

因此，我们可以说，通过内部的中间件，我们把执行顺序改为：

(1) `roleFactory('admin')`

(2) `app.get('/')`

如果想在检查授权时多增加一些角色呢？可以把路由改为：

```
var powerUsers = [roleFactory('admin'),roleFactory('client')];
app.get('/', powerUsers, function(req, res) {
  res.send('Welcome to Express!');
});
```

因为传入了一组中间件，所以就把页面执行限制成了同时是 admin 和 client 角色的用户，并把执行顺序改为了：

(1) `roleFactory('admin')`

(2) `roleFactory('client')`

(3) `app.get('/')`

因为每个 `roleFactory` 方法都要求对应的角色在 session 中出现，所以用户必须同时是 client 和 admin 才能访问该页面。

7.2 Socket.IO

Socket.IO 是一个小巧的扩展库，功能很像 Node 的核心库 `net`。你可以通过 Socket.IO 在浏览器客户端与 Node 服务器之间采用高效的底层 socket 机制来回发送消息。该模块的另一个优点是，可以在浏览器与服务器间共享代码。也就是说，一旦你建立了连接，就可以用同样的 JavaScript 代码在两边进行消息通信。

Socket.IO 的名字源于它使用了浏览器支持并采用的 HTML5 WebSocket 标准。幸运的是，该库还支持一系列降级功能：

- WebSocket
- WebSocket over Flash
- XHR Polling
- XHR Multipart Streaming
- Forever Iframe
- JSONP Polling

在大部分情景下，你都能通过这些功能选择与浏览器保持类似长连接的功能。Socket.IO 模块使用同一套 API，让连接服务器和浏览器的代码能够共享连接代码。

创建 Socket.IO 实例很简单，只要包含该模块并创建服务器对象就行。使用 Socket.IO 还有一点小区别，就是它还需要依赖 HTTP 服务器，见例 7-16。

例 7-16　创建 Socket.IO 服务器

```
    var http = require('http'),
    io = require('socket.io');

server = http.createServer();
server.on('request', function(req, res){
  // 常见的 HTTP 服务器内容
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World');
});

server.listen(80);
```

```
var socket = io.listen(server);

socket.on('connection', function(client){
  console.log('Client connected');
});
```

本例子中的 HTTP 服务器可以做任何事情。在这个例子里，我们只是让它返回“Hello World.”。其实，Socket.IO 并不关心 HTTP 服务器做什么，它只是把自带的事件监听器包装在发送到服务器的所有请求上，该监听器会查找从 Socket.IO 客户端发送来的请求，并对应处理。对于其他的请求，它会以原本的工作方式传递给 HTTP 服务器。

本例子调用监听类的工厂方法 `io.listen()` 创建了一个 socket.io 服务器。因为 socket 是持久性连接，你不需要像 HTTP 服务器那样处理 `req` 和 `res` 对象。与使用 `net` 类似，你需要使用传入的 `client` 对象来与每个浏览器进行通信。当然，在浏览器里也需要放置一些代码（例 7-17）来与服务器交互，这同样重要。

例 7-17　与 Socket.IO 服务器交互的小网页

```
<!DOCTYPE html>
<html>
  <body>
    <script src="/socket.io/socket.io.js"></script>
    <script>
      var socket = io.connect('http://localhost:8080');
      socket.on('message', function(data){ console.log(data) })
    </script>
  </body>
</html>
```

这个简单的页面做的事情是从 Node 服务器（localhost 的 8080 端口）直接加载需要的 Socket.IO 客户端库。

> 虽然 80 端口是 HTTP 的标准端口，但 8080 端口在开发的时候更为方便，因为许多开发者是在本地机器上允许 Web 服务器进行测试的，这样会和 Node 的功能冲突。而且，许多 Linux 系统内建的安全规则不允许非管理员用户使用 80 端口，所以使用一个较大的数字作为端口更为方便。

接下来，我们为准备连接的 Socket.IO 服务器名字创建一个新的 `Socket` 对象。然后调用 `socket.connect()` 进行连接，并为消息事件添加了一个监听器。每当服务器给这个客户端发送一条消息，客户端就会把它输出到浏览器的终端窗口上。

现在，让我们修改一下服务器的代码，让它把此页面发送给客户端，并测试一下

（例 7-18）。

例 7-18　一个简单的 Socket.IO 服务器

```
    var http = require('http'),
    io = require('socket.io'),
    fs = require('fs');

var sockFile = fs.readFileSync('socket.html');

server = http.createServer();
server.on('request', function(req, res){
  res.writeHead(200, {'content-type': 'text/html'});
  res.end(sockFile);
});

server.listen(8080);

var socket = io.listen(server);

socket.on('connection', function(client){
  console.log('Client connected');
  client.send('Welcome client ' + client.sessionId);
});
```

本例子中最大的变化是增加了 `fs.readFileSync` 函数，它把外部的网页文件带入了 socket 服务器。现在浏览器的请求不再是返回 Hello World 了，Node 服务器会返回 socket.html 的内容。因为 `readFileSync` 是一个同步函数，所以它会堵塞 Node 的事件循环，直到文件读取完毕。这样能确保当服务器准备好接受连接的时候，客户端能够马上得到文件的内容。

现在每当有人向服务器发起任意请求（除非是直接往 Socket.IO 客户端库发起的请求），他都会得到 socket.html 的一份拷贝（如例 7-17 中的代码）。连接的回调函数被扩展成往客户端发送一条欢迎信息，然后例 7-18 中的客户端会在它的终端上显示如 `Welcome client 17844937089830637` 这样的内容，其中的数字 ID 是用 `Math.random()` 生成的一个整数。

7.2.1　命名空间

当你能够完全控制自己的程序与架构时，可以如例子中的方法那样创建 websocket。但当你把它们添加到已在使用 socket 的程序，或是你正在写的服务需要嵌入到别人的项目中去时，这很容易会导致冲突。例 7-19 演示了如何用命名空间把 Socket.IO 的监听器有效地区分到频道中，从而避免类似问题。

例 7-19　修改网页来使用 Socket.IO 的命名空间

```
<!DOCTYPE html>
```

```
<html>
  <body>
    <script src="/socket.io/socket.io.js"></script>
    <script>
      var upandrunning = io.connect('http://localhost:8080/
upandrunning');
      var weather = io.connect('http://localhost:8080/weather');
      upandrunning.on('message', function(data){
        document.write('<br /><br />Node: Up and Running Update<br />');
        document.write(data);
      });
      weather.on('message', function(data){
        document.write('<br /><br />Weather Update<br />');
        document.write(data);
      });
    </script>
  </body>
</html>
```

这个更新后的 socket.html 版本创建了两个 Socket.IO 连接，一个是 http://localhost:8080/upandrunning，一个是 http://localhost:8080/weather。每个连接有自己独立的变量和 `.on()` 事件监听器。除了以上的区别，Socket.IO 工作的方式保持不变。例 7-20 更改了在终端记录日志的方法，改为直接在 Web 浏览器窗口中显示消息结果。

例 7-20　使用空间的 Socket.IO 服务器

```
var sockFile = fs.readFileSync('socket.html');

server = http.createServer();
server.on('request', function(req, res){
  res.writeHead(200, {'content-type': 'text/html'});
  res.end(sockFile);
});

server.listen(8080);

var socket = io.listen(server);

socket.of('/upandrunning')
  .on('connection', function(client){
    console.log('Client connected to Up and Running namespace.');
    client.send("Welcome to 'Up and Running'");
});

socket.of('/weather')
  .on('connection', function(client){
    console.log('Client connected to Weather namespace.');
    client.send("Welcome to 'Weather Updates'");
});
```

`socket.of` 函数把 socket 对象切分成多个独立的命名空间，每个空间有自己的处理规则。如果一个客户端连接到 http://localhost:8080/weather 并发起 `emit()` 命令，它

的结果只会在该命名空间里被处理，而不会在 /upandrunning 命名空间里处理。

7.2.2 Express中使用Socket.IO

许多时候你会想要使用 Socket.IO，把它作为 Node 中的独立程序或者一个包含 Node 之外模块的大型网站的一部分。如果是同时使用 Express 和 Socket.IO，你将会在使用统一的语言（JavaScript）编写整个软件的结构（包括面向客户端的视图）方面获得巨大便利。

下面，先把例 7-21 保存为 socket_express.html。

例 7-21　Socket.IO 绑定到 Express 应用上：客户端代码

```
<script src="/socket.io/socket.io.js"></script>
<script>
var socket = io.connect('http://localhost:8080');
socket.on('news', function(data) {
  document.write('<h1>' + data.title + '</h1>' );
  document.write('<p>' + data.contents + '</p>' );
  if ( data.allowResponse ) {
    socket.emit('scoop', { contents: 'News data received by client.' });
  }
});
</script>
```

本例子从连接 Socket.IO 的 8080 端口开始。当 Socket.IO 服务器发送 news 事件的时候，客户端会把新项目的标题和内容写到浏览器页面上。如果该 news 项目允许有反馈，客户端 socket 还会发起 scoop 事件。scoop 对于报道源没有什么用处，它只是表示客户端反馈接收了原始新闻。

作为一个新闻业务的例子，服务器对 scoop 事件的响应是发起另一个新闻消息，客户端会收到新的事件并打印到屏幕上。为了防止这一循环无休止进行下去，发送新闻消息的时候会同时发送 `allowResponse` 参数。如果它是 false 或完全没有出现（见例 7-22），客户端则停止发送 scoop。

例 7-22 演示了如何结合 Express 服务器使用。

例 7-22　Socket.IO 绑定到 Express 应用上：服务器代码

```
var app = require('express').createServer(),
     io = require('socket.io').listen(app);

app.listen(8080);

app.get('/', function(req,res) {
  res.sendfile(__dirname + '/socket_express.html');
});
```

```
io.sockets.on('connection', function(socket) {
  socket.emit('news', {
    title: 'Welcome to World News',
    contents: 'This news flash was sent from Node.js!',
    allowResponse: true
  });
  socket.on('scoop', function(data) {
    socket.emit('news', {
      title: 'Circular Emissions Worked',
      contents: 'Received this content: ' + data.contents
    });
  });
});
```

先创建 Express 服务器并处理，然后把它作为参数传入 Socket.IO。 当 Express 应用从 `listen()` 函数开始运行后，Web 服务器和 socket 服务器同时启动。下一步，定义在根路径（/）上的路由负责把例 7-21 创建的客户端文件发送出去。

新闻中心的服务器代码与客户端代码看起来很像，这是有原因的。在 Node 和 Web 浏览器中同样的事件（emit、on 消息、connection）行为类似，这使得连接的获得更加直观。因为数据是作为 JavaScript 对象在两边传递的，所以不需要额外的解析或序列化工作。

可见，通过把 Socket.IO 加入 Express，我们可以立马获得许多功能，但精明的程序员会发现这种单向的通信方式价值有限，除非从用户浏览器发起的连接能够以 socket 流的方式使用。任何修改（登出、修改设置等操作）应该在 socket 操作中反馈出来，反之亦然。如何完成此功能呢？答案是使用 session。

接下来演示一下如何用 session 来做权限验证，先看一下客户端代码（views/socket.html），见例 7-23。

例 7-23　客户端 HTML（Jade 模板）: Socket.IO sessions

```
!!! 5
html(lang='en')
  head
    script(type='text/javascript', src='/socket.io/socket.io.js')
    script(type='text/javascript')
      var socket = io.connect('http://localhost:8080');
      socket.on('emailchanged', function(data) {
        document.getElementById('email').value = data.email;
      });
      var submitEmail = function(form) {
        socket.emit('emailupdate', {email: form.email.value});
        return false;
      };
  body
```

```
h1 Welcome!

form(onsubmit='return submitEmail(this);')
  input(id='email', name='email', type='text', value=locals.email)
  input(type='submit', value='Change Email')
```

当浏览器渲染时，这个页面会显示一个表单文本框（内容为 Change Email），默认值是从 Express 的 session 数据中的 `locals.email` 变量取得。用户输入后，应用程序进行以下操作：

(1) 创建一个 Socket.IO 连接，并把用户的所有 Email 更改以 `emailupdate` 事件发送出去。

(2) 监听 `emailchanged` 事件，当服务器返回新的 Email 地址时，更改文本框的内容（后续我们会讲解更多这方面的内容）。

接下来，我们看看例 7-24 的 Node.js 代码部分。

例 7-24　在 Express 和 Socket.IO 间共享 session 数据

```
var io = require('socket.io');
var express = require('express');
var app = express.createServer();
var store = new express.session.MemoryStore;
var utils = require('connect').utils;
var Session = require('connect').middleware.session.Session;

app.configure(function() {
  app.use(express.cookieParser());
  app.use(express.session({secret: 'secretKey', key: 'express.sid',
store: store}));
  app.use(function(req, res) {
    var sess = req.session;
    res.render('socket.jade', {
      email: sess.email || ''
    });
  });
});

// 启动应用
app.listen(8080);

var sio = io.listen(app);

sio.configure(function() {
  sio.set('authorization', function (data, accept ) {
    var cookies = utils.parseCookie(data.headers.cookie);
    data.sessionID = cookies['express.sid'];
    data.sessionStore = store;
    store.get(data.sessionID, function(err, session) {
      if ( err || !session ) {
```

```
        return accept("Invalid session", false);
      }
      data.session = new Session(data, session);
      accept(null,true);
    });
  });

  sio.sockets.on('connection', function(socket) {
    var session = socket.handshake.session;
    socket.join(socket.handshake.sessionId);
    socket.on('emailupdate', function(data) {
      session.email = data.email;
      session.save();
      sio.sockets.in(socket.handshake.sessionId).emit('emailchanged', {
        email: data.email
      });
    });
  });
});
```

这个例子使用了 Connect 的中间件框架来简化公共操作，比如 session 管理、cookies 操作、用户认证、缓存、性能指标等。在本例子中，cookie 和 session 工具用来处理用户数据。Socket.IO 并不知道 Express 的存在，反之也如此，所以 Socket.IO 并不知道当用户连接时的 session。但是所有模块都需要使用 Session 对象来共享数据，这很好地演示了概念分离（Separation of Concerns，SoC）[4] 的编程范例。

这个例子演示了在连接之后，通过解析用户头信息，用户认证后使用 Socket.IO。因为 session 的 ID 是通过 cookie 传给服务器的，所以你可以通过此值来读取 Express 的 session ID。

这次，Express 设置包含了一行 session 管理配置。创建 session 管理器有几个参数，分别是 secret（用来防止 session 破解）、key（在 Web 浏览器 cookie 中用来保存 session ID）、store 对象（用来保存 session 数据，以便后续获取），其中 store 对象是最重要的。这个例子是自己创建一个变量，然后传给 Express，而不是由 Express 创建和管理内存存储。现在 session 存储可以跨越 Express，让整个应用都能访问了。

下一步，为默认（/）网页创建一个路由。在之前的 Socket.IO 例子中，这个函数是用来直接把 HTML 输出到 Web 浏览器的。这次，Express 会把 views/socket.jade 的内容渲染以后再发给 Web 浏览器。`render()` 的第二个变量是保存在 session 里的 E-mail 地址，在例 7-23 中将会把它解析并作为文本框的默认值。

注 4：SoC 指的是把软件拆分成更小的独立模块（关注点），尽量使它们互相之间没有重叠功能。中间件有助于这样的设计风格，因为它能允许完全独立的模块在一个公共环境中交互，而且不需要知道彼此的情况。但是，正如我们所见的如 bodyParser() 这样的模块，它还是需要程序员来了解模块间最终是如何交互的，并由程序员按合适的顺序和上下文情况来使用它。

真正的动作是在 Socket.IO 的 `'authorization'` 事件中完成的。当 Web 浏览器连接到服务器时，Socket.IO 进行权限验证操作，以确认该连接是否可以进行后续操作。这个例子中的验证标准是 session 是否有效，这是当用户读取网页时由 Express 提供的。Socket.IO 通过 `parseCookie`（Connect 框架的功能）从请求头中读取 session ID，然后从内存存储中读取 session，接着根据接收的信息创建 `Session` 对象。

传递给 authorization 事件的数据是保存在 socket 的握手（`handshake`）资源中。因此，在数据对象中保存 session 对象使得它能够在 socket 的生命周期中使用。当创建 `Session` 对象时，使用创建的内存存储并传递给 Express，这样 Express 和 Socket.IO 都可以访问同样的 session 数据。Express 通过 `req.session` 对象访问，sockets 通过 `socket.handshake.session` 对象访问。

假设一切运行顺利，调用 `accept()` 对 socket 进行授权，并允许该连接继续运行。

现在可以支持用户在浏览器中从不同的标签页面同时访问你的网站了。这样同一个 session 会创建两个连接，那么你该如何处理已连接的 socket 的更新呢？ Socket.IO 提供了*房间*（room）和*频道*（channel），你可以根据自己的喜好使用它们。在例 7-24 中，通过用 `sessionId` 作为参数来初始化 `join()` 命令，socket 透明地创建了一个专用频道，这样你就可以往该用户使用的所有连接中发送消息。登出是这一技术的明显应用。当用户从一个标签页中登出时，该登出命令会立即传送给其他所有页面，让用户所看见的该应用页面的状态保持一致。

在修改 session 数据后，确保记得执行 `session.save()`，否则该修改不会反映在后续的请求中。

第 8 章

扩展Node

8.1　模块

Node 的模块系统使其很容易创建扩展功能。它很好学习，并能让我们轻松编写可重用的代码库。Node 模块系统基于 commonJS 模块标准。在前面的章节中，我们已经使用了许多模块了，现在我们学习如何创建自己的模块。例 8-1 演示了一个简单的实现。

例 8-1　一个简单的模块

```
exports.myMethod = function() { console.log('Method output') };
exports.property = "blue";
```

正如你所见，编写一个模块就是往全局变量 `exports` 上添加一些属性那么简单。任何通过 `require()` 包含进来的脚本都会返回它的 `exports` 对象，这意味着所有从 `require()` 返回的内容都在一个闭包里，你在模块内使用的私有变量并不会暴露给整个程序的作用域。

Node 开发人员已经为模块创建了一些约定。首先，它一般通过创建工厂方法来构建类。虽然也可以暴露类本身，但工厂方法给了我们一种整洁的方法来实例化对象。对于 I/O 相关的类，其中一个参数通常是个回调函数，或者用来表示 I/O 操作完成，或者表示其中一些常见的环节。比如，`http.Server` 有一个工厂方法叫 `http.createServer()`，它会对 `http.Server` 中最常见的 `request` 事件调用回调函数。

8.2 包管理

可以自己编写模块固然很好，但最终还需要有好的方法来发布它们，并与团队的其他成员或与社区进行分享。Node 的包管理系统（npm）提供了发布代码的方法，或者将代码发布到本地，或者通过全局的 Node 模块资源库。npm 帮助你管理代码依赖包的安装，以及其他与发布代码相关的工作。而且，npm 是完全用 JavaScript 和 Node 编写的，所以你如果已经在使用 Node，那么就已经在使用 npm 了。npm 给开发者提供了安装工具，也给模块维护人员提供了发布工具。

大部分开发人员开始使用 npm 是通过简单的 `npm install` 命令来安装模块包。你可以安装自己本地下载的包，但更多时候需要用 npm 在注册库中安装远程的包。注册库保存了其他 Node 开发者共享的包，供你使用，如数据库驱动、流控制库、数学库。你用 npm 安装的大部分库是完全用 JavaScript 编写的，但也有少数需要编译。幸运的是，npm 会帮你完成这些工作。你可以在 http://search.npmjs.org 上查看注册库中有哪些内容。

8.2.1 搜索包

`search` 命令会列出在全局 npm 注册库中的所有包，并根据包名进行过滤：

```
npm search packagename
```

如果你没有提供包的名字，那么所有可用的包都会显示出来。

如果包列表过时了（因为添加或删除了某个包，或者你知道某个需要的包并没有出现），可以用以下命令操作 npm 来清理缓存：

```
npm cache clean
```

下次你问 npm 要包列表的时候，因为需要重建缓存，所以该命令的执行时间会长一些。

8.2.2 创建包

虽然你用 npm 命令安装的大部分包也能为其他使用 Node 的用户所用，但编写一个包并不需要对外公开。把你自己的代码集中在模块包里，有利于在不同项目中复用、与其他开发者共享，或者是在演示或生产线服务器上运行你的程序。

包并不限于模块或是扩展库，在许多情况下，包可以包含整个需要部署的应用。把文件都打成包能使部署更加简单，可以声明好依赖关系，解决从开发环境部署到生产环境时通常需要猜测依赖库的难题。

创建一个包并不需要很多额外的工作，只需要创建一个 package.json 文件，并添加关于你的模块的简单说明（包的名字和版本号是最重要的部分）。要快速生成一个有效的包文件，可以在你的模块文件夹路径下运行命令 `npm init`，它会提示你输入关于该模块的描述信息，然后该命令会在当前目录下生成一个 packages.json 文件。如果已经存在包文件，它的属性会被作为默认值，并允许修改。

要使用自己的包，可以使用命令 `npm install /path/to/yourpacage` 进行安装。路径可以是本地文件系统的一个文件夹，也可以是一个外部 URL（比如 GitHub）。

8.2.3 发布包

如果你的模块对大众用户也有用，并且已经准备好了，那么可以通过 `npm` 的 `publish` 命令对外发布。发布包内容需要以下操作。

(1) 用 `adduser` 命令创建一个用户：

```
npm adduser
```

依照出现的提示操作，输入用户名、密码和邮箱地址。

(2) 用 `publish` 命令发布包：

```
npm publish
```

这些就是需要操作的所有内容。目前，并不需要注册或者验证有效性。

> 这体现了关于 npm 的一个值得关注的特性：因为所有人都可以发布包，而且没有进行过滤或监督，所以你用 npm 安装的包的质量是没有保证的。所以说，用者自理。

如果之后想要取消发布自己的包，可以通过 `npm unpublish` 命令实现。注意，你需要清理自己的包列表缓存。

8.2.4 链接

虽然 `npm` 擅长发布和部署，但当初设计它的目的主要是作为开发过程中管理依赖项的工具。`npm link` 命令能创建你的项目及其依赖项之间的符号链接，因此在项目开发过程中，依赖项中的任何改动都能为你所用。

实现此效果之所以必要，有如下两个主要原因。

- 你想从当前项目中通过 `require()` 访问另外一个项目中的功能。

- 你想在多个项目中使用同一个包，而且不需要在每个项目中都维护一个版本。

不加参数的情况下调用 `npm link` 会为当前项目在全局包路径中创建一个符号链接，让你系统上的其他所有项目都能使用。要使用这个功能，正如前面所说，你需要有一个 package.json 文件。利用 `npm init` 是生成这个文件的初始版本的最快方法。

输入 `npm link packagename` 会为该包创建一个从当前项目工作目录到全局模块路径中的符号链接。比如，输入 `npm link express` 会在全局包路径中安装 Express 框架，并包含到你的项目中来。每当 Express 升级，你的项目会自动从全局包文件夹中找到最新版本来使用。如果你在多个项目中链接了 Express，所有这些项目都会同步到最新的版本，这样就不需要在 Express 升级时一个一个地操作了。

8.3　附加组件

模块是指用 JavaScript 编写的 Node 扩展，而*附加组件*是指用 C/C++ 编写的扩展。附加组件通常是把现有的系统库包装起来，然后把它们的功能暴露给 Node 使用。当然，也可以用它们来创建新的功能，但许多人显然会选择使用 JavaScript 来做这样的事情。具体来说，附加组件是动态链接的共享的对象。

要创建一个附加组件，至少需要两组文件：组件的代码和编译好的文件。Node 采用的是 Python 的 `waf` 编译系统。让我们先从 Hello World 例子入手吧。例 8-2 的功能与在 JavaScript 里写上 `exports.hello = "world";` 是等价的。

例 8-2　一个简单的 Node 附加组件

```
#include <v8.h>

using namespace v8;

extern "C" void init (Handle<Object> target) {
  HandleScope scope;
  target->Set(String::New("hello"), String::New("world"));
}
```

这段代码做的第一件事情是把 `v8` 的头文件包含进来，因为 Node 是构建在 V8 基础上的。它提供了我们将要用到的许多标准对象。下一步，我们声明了命名空间，然后创建了*封装器*，所有附加组件都需要这样操作。封装函数就像 JavaScript 模块中的全局 `exports` 变量那样。我们会将需要从附加组件暴露出来的内容通过 `extern 'C' void init(Handle<Object> target)` 这个函数对外公开。

词汇表

阻塞操作

阻塞操作在长时间等待资源的时候，要求程序暂停下来。

通常是请求硬件资源（如磁盘）或网络资源（如 HTTP 请求）。因为请求无法马上从该长时间运行的资源中获得结果，所以即使这时候系统的 CPU 和内存有空闲，后续的操作也都会被堵塞，直到该请求完成后才能继续运行。

回调

回调函数是指当一个堵塞操作完成之后会被“调用”的函数，通常指的是磁盘访问这类 I/O 操作。回调函数可以带参数使用。

类

见“伪类”。

函数

通过一组变量调用的一段代码集合，可以返回唯一的一个值。在 JavaScript 中，函数也有上下文关系，因此 `this` 变量是保留值。JavaScript 中的函数被认为是第一类（first class），因此也可以把它们当成变量或对象的属性。

方法

对象的功能单元。

参见“函数”。

非阻塞操作

非阻塞操作不会堵塞别人。

参见“阻塞操作”。

伪类

伪类是在 JavaScript 中创建抽象对象的方法，目的是为了以后初始化成为对象。可以通过 `new` 方法把伪类转变成对象。为了与其他类型对象区别开，伪类的首字母采用大写形式。比如，`Server` 是一个伪类，而 `server` 则可能是它的一个实例。

索引

关于作者

Tom Hughes-Croucher 是一名程序员，同时也是技术布道师。他曾效力于许多响当当的大公司，或与他们合作，如 Yahoo!、NASA、Tesco、Walmart、MySpace、Three Telecom 以及 UK Channel 4。Tom 向万维网联盟（W3C）和英国标准协会（BSI）提交了多项网络标准提案。

Mike Wilson 曾与世界一流大品牌公司合作，这些公司包括 Disney、Microsoft 和 McDonald。他有多年网络开发经验，从小企业网站到百万用户在线的大型 MMO 服务器集群都曾设计并构建过。在闲暇时间，Mike 会更新他的个人博客，在论坛上发文章，以及尝试新的框架和软件。目前，他与太太及三个孩子一起住在温哥华。

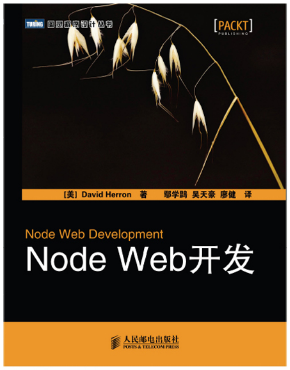
关于封面

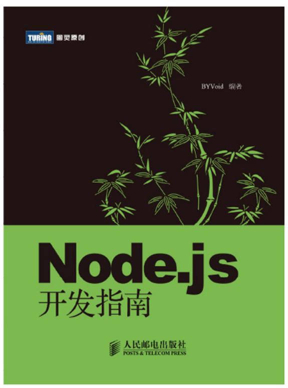
本书封面上的动物是普通树鼩（Tupaia glis），树栖哺乳动物，出没在东南亚的南部地区，通常生活在森林中，偶尔也会出现在果园和花园里。它们善于攀爬，并能跳跃跨过两英尺远的树木。它们白天活动，以植物、种子、水果、蚂蚁、蜘蛛和小蜥蜴为食。

普通树鼩身长 6~8 英寸，毛茸茸的尾巴和身子一样长，嘴尖细，爪有五趾，皮毛呈黑色、灰色或淡红色，肚皮呈白色。“树鼩”这个属名源于马来语的“松鼠”，因为两者有些相似。人们曾经认为树鼩与灵长类动物沾亲带故，但现在它们有了自己的目——树鼩目。

普通树鼩在几个月大的时候就会性成熟，然后进行交配。雄性树鼩会筑造两个巢穴，一个给自己与配偶，一个给幼仔。但是父母对幼仔关爱不足，雌性树鼩两天才去看望幼仔一次，仅哺育它们几分钟。

封面图片出自 Lydekker 所著的 Natural History 一书。


[PACKT]
[美] David Herron 著 鄢学鹍 吴天豪 廖健 译
Node Web Development
Node Web开发
人民邮电出版社
POSTS & TELECOM PRESS


BYVoid 编著
Node.js
开发指南
人民邮电出版社
POSTS & TELECOM PRESS


Professional JavaScript for Web Developers 3rd Edition
JavaScript
高级程序设计
（第3版）
[美] Nicholas C.Zakas 著
李松峰 曹力 译
人民邮电出版社
POSTS & TELECOM PRESS

Node即学即用

Node正迅速成为Web开发社区里最有影响力的技术之一。你一定想快速掌握Node，学习如何用JavaScript开发服务器程序。有了这本指南，你就能学会用Node构建高度可扩展的服务器程序，理解它的事件循环架构如何降低开发的复杂度并且保证服务器编程的安全与便捷。

本书是Node开源框架主要贡献者的最新力作，解析了为什么Node的单线程方法能够在多台服务器间支撑起大量的并发连接，并让我们看到了在浏览器与服务器间共享代码是何等便利。Node何以能俘获Google、LinkedIn及eBay等众多大牌公司的芳心？本书将向你解释其原委。

通过阅读本书，你可以：

- 学习Node的事件循环架构、非阻塞I/O和事件驱动编程模型；
- 动手编写I/O示例应用，其中包括一个聊天服务器；
- 用现成的设计模式编写事件驱动程序；
- 在多核环境下高效地运用Node的单线程策略；
- 配合具体例子，深入框架核心及API工具；
- 学习Node如何支持多种数据库和存储工具；
- 利用Node庞大的模块库构建新的扩展。

“本书探讨了Node及许多第三方模块，并给出了指导练习，旨在带你了解Node。通过学习本书，你不但能够熟悉JavaScript的基本操作，还能逐渐开始构建复杂、交互式的网站。如果你曾经使用过其他服务器端Web框架，定会震惊于用Node这么容易就能编写一个服务器！”

——Ryan Dahl，Node之父

“本书很好地诠释了Node的精髓，并讲述了如何用它构建交互式网络应用和网站。Node棒极了，而本书就是关于Node的很好的指南，请尽情享受阅读的乐趣吧！”

——Brendan Eich，JavaScript之父

封面设计：Karen Montgomery　张健

图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604

分类建议　计算机/Web开发

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

O'REILLY®
oreilly.com.cn

ISBN 978-7-115-30618-0

ISBN 978-7-115-30618-0
定价：39.00元